Georgia State University
ScholarWorks @ Georgia State University

Computer Science Dissertations | Department of Computer Science

Spring 5-10-2013

Scientific High Performance Computing (HPC) Applications On The Azure Cloud Platform

Dinesh Agarwal

Follow this and additional works at: https://scholarworks.gsu.edu/cs_diss

Recommended Citation
Agarwal, Dinesh, "Scientific High Performance Computing (HPC) Applications On The Azure Cloud Platform." Dissertation, Georgia State University, 2013.
https://scholarworks.gsu.edu/cs_diss/75

This Dissertation is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Dissertations by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

TITLE:

SCIENTIFIC HIGH PERFORMANCE COMPUTING (HPC) APPLICATIONS ON THE AZURE CLOUD PLATFORM

by

DINESH AGARWAL

Under the Direction of Dr. Sushil K. Prasad

ABSTRACT

Cloud computing is emerging as a promising platform for compute and data intensive scientific applications. Thanks to the on-demand elastic provisioning capabilities, cloud computing has instigated curiosity among researchers from a wide range of disciplines. However, even though many vendors have rolled out their commercial cloud infrastructures, the service offerings are usually only best-effort based without any performance guarantees. Utilization of these resources will be questionable if it can not meet the performance expectations of deployed applications. Additionally, the lack of the familiar development tools hamper the

productivity of eScience developers to write robust scientific high performance computing (HPC) applications. There are no standard frameworks that are currently supported by any large set of vendors offering cloud computing services. Consequently, the application portability among different cloud platforms for scientific applications is hard. Among all clouds, the emerging Azure cloud from Microsoft in particular remains a challenge for HPC program development both due to lack of its support for traditional parallel programming support such as Message Passing Interface (MPI) and map-reduce and due to its evolving application programming interfaces (APIs). We have designed newer frameworks and runtime environments to help HPC application developers by providing them with easy to use tools similar to those known from traditional parallel and distributed computing environment setting, such as MPI, for scientific application development on the Azure cloud platform. It is challenging to create an efficient framework for any cloud platform, including the Windows Azure platform, as they are mostly offered to users as a black-box with a set of application programming interfaces (APIs) to access various service components. The primary contributions of this Ph.D. thesis are (i) creating a generic framework for bag-of-tasks HPC applications to serve as the basic building block for application development on the Azure cloud platform, (ii) creating a set of APIs for HPC application development over the Azure cloud platform, which is similar to message passing interface (MPI) from traditional parallel and distributed setting, and (iii) implementing *Crayons* using the proposed APIs as the first end-to-end parallel scientific application to parallelize the fundamental GIS operations.

INDEX WORDS: Cloud computing, GIS computations using cloud platforms, Windows Azure cloud platform,Scientific applications over cloud platforms

TITLE: SCIENTIFIC HIGH PERFORMANCE COMPUTING (HPC) APPLICATIONS ON THE AZURE CLOUD PLATFORM

by

DINESH AGARWAL

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2013

Copyright by
Dinesh Agarwal
2013

TITLE: SCIENTIFIC HIGH PERFORMANCE COMPUTING (HPC) APPLICATIONS ON THE AZURE CLOUD PLATFORM

by

DINESH AGARWAL

Committee Chair: Dr. Sushil K. Prasad

Committee: Dr. Yi Pan

Dr. Xiaolin Hu

Dr. Shamkant Navathe

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2013

DEDICATION

For their love and support, I dedicate this thesis to my parents.

ACKNOWLEDGEMENTS

I owe a great deal of gratitude to my advisors, colleagues, and friends for the guidance, support, and motivation they have blessed me with. I wish to express my sincere thanks to them.

I would like to thank my advisor, Prof. Sushil Prasad for his untiring support and guidance throughout this work. Without his insightful research advice, constant encouragement, generous support, and patience when research did not progress as expected, I would not have been able to finish this degree. He was always there standing by my side guiding me towards the right path on my pursuit of Ph.D. I cannot thank him enough for his belief in me that kept me motivated.

I would like to thank my entire committee: Prof. Shamkant Navathe, Dr. Xiaolin Hu, and Prof. Yi Pan for their help and guidance throughout the research. I offer them my sincere gratitude for their invaluable feedback on my research.

I cannot thank Prof. Rajshekhar Sunderraman enough for his help and guidance on the academic as well as administrative aspects of this Ph.D. I am also indebted to him for introducing me to the game of Squash that has been my stress buster since a long time now.

I am grateful to Prof. Sanjay Chaudhary and Dr. Abhinay Pandya for encouraging me to follow the path to higher education and helping me to start my career as a Ph.D. student. If it was not for them, I would not have attained the intellectual satisfaction that I enjoy today.

I thank my colleagues Satish Puri, Xi He, Thamer Mohsen, Rasanjalee Dissanayaka, Chad Frederick, Nick Mancuso, John Daigle and other members of the DiMoS Lab for the intellectual discussions we had and the camaraderie that we shared.

I have been humbled by the service and support of the staff at the Department of Computer Science. I would like to mention Mr. Shaochieh Ou for his help with all small and large challenges I faced working with as small as single desktop machines to large compute-

clusters. I cannot forget to mention Ms. Tammie Dudley for her kind and helpful nature.

I would not have been able to come this far in life without the love, support, and blessings of my loved ones. They deserve much credit for my accomplishments. My father, Mr. Hanuman Prasad Agarwal has always inspired me, by example, to work hard and persist. Starting from humble beginnings what he has accomplished in life has always kept me motivated to think big and not to be afraid to try it. The regular video conference calls with my mother, Mrs. Saraswati Agarwal, since I came to the United States, have kept me motivated to achieve higher goals in my life. I also thank her for the never ending supplies of home-made snacks at my place. I extend my gratitude to my brother, Mukesh Agarwal for always being the understanding elder brother and a friend who always knew the right words to say. I am equally thankful to his wife, Shalini for her love and affection that I have been pampered with. I must mention my niece and nephew Rinkle and Moulik for taking away my stress with their innocence and beautiful smiles.

I am thankful to Sarang Sunder for being a good friend and room mate to me. Archana Kath and Riddhi Shah have been the best neighbors one can get and I extend my sincere gratitude towards them. I am going to cherish the memories of our experiments in the kitchen for the rest of my life.

Finally, I am thankful to the Molecular Basis of Disease group for their approval of my worthiness by extending me the MBD fellowship for the last few years of my stay at GSU.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PART 1

INTRODUCTION

Cloud computing provides users with a quick access to large-scale compute and storage resources without having to worry about the setup, maintenance, or initial investment. The utility based framework facilitates experimenting with large amount of compute power while obviating the need to own a parallel or distributed system[3]. Scientific applications with varying load cycles are the perfect fit for the pay-as-you-go cost model of cloud computing as the resources in cloud platform can be allocated and deallocated on-demand based on the application's requirements.

Cloud computing promises scientists with a new infrastructure and paradigm for large-scale distributed computing [4]. The emerging cloud platforms, such as Microsoft's Azure—with their potential for large-scale computing and storage capabilities, easy accessibility by common users and scientists, on demand availability, easy maintenance, sustainability, and portability—have the promise to be the platform of choice for a wide range of scientific applications.

The term cloud computing started getting popular in 2007 and in terms of search trends—as reported by Google search trends in Figure 1.1—it has been consistently more popular than grid computing since late 2008. The initial distinction between grid computing and cloud computing was so hazy that in a report on Amazon's cloud services, published by Garfinkel [5], the author titles these services as grid computing. As a matter of fact, there is no concise definition of cloud computing yet [3, 6–9].

Our review of literature points to the fact that a large section of the academic community debates if cloud computing is a new paradigm [3, 8, 10] or just grid computing in a new wrapper. Buyya et al.[10] argues that the cloud computing appears to be similar to the grid computing only at a cursory glance but a closer observation presents a different

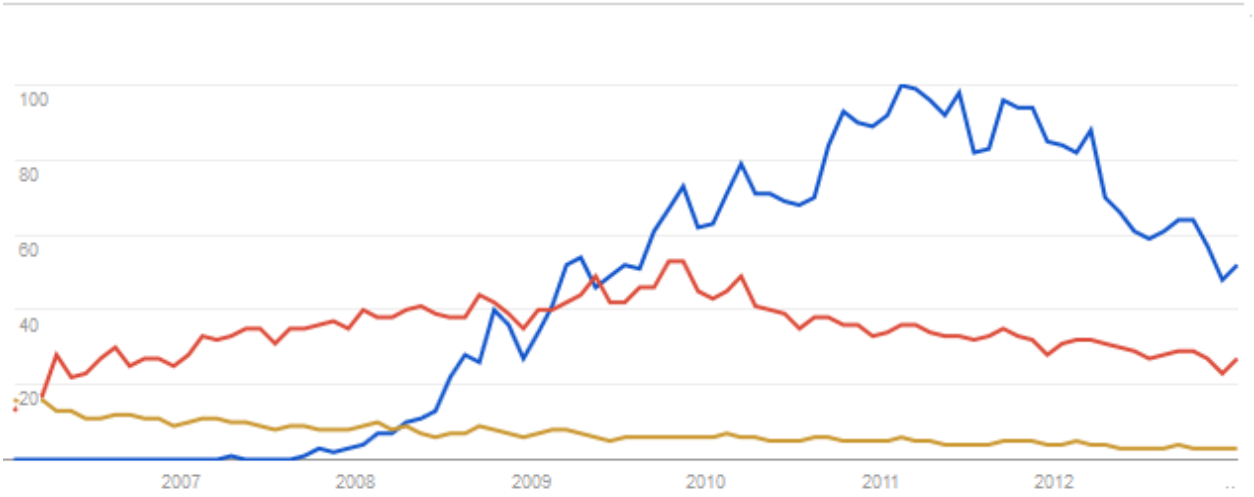


Figure 1.1 Google search trends for cloud computing (blue line), Grid computing (yellow line), and Virtualization technology (red line).

case. Armburst et al.[3] support the claim of Buyya et al. adding that the cloud computing platform uniquely provides an illusion of infinite available resources. Lee [4] advocates the difference by employing the case of hurricane Katrina in 2005 to conclude that the only answer to the *scientific and operational grand challenge problem* is enormous computer power. However, it is not economically possible to dedicate the required amount of resources for this single purpose. Therefore (i) resources must be shared but available on-demand, (ii) the platform should be scalable on-demand, and (iii) resources should be easily accessible in a user friendly way over the web. The grid computing platform or any other large compute cluster cannot adapt to these guidelines.

Foster et. al[8] present a comprehensive comparison of the grid computing and the cloud computing platforms. The authors recognize the similarity in the two platforms in terms of the vision and challenges, but the authors also make a solid case to differentiate the two platforms in terms of scale of operation. The authors agree that the more massive scale being offered by the cloud computing platform can demand fundamentally different approaches to tackle a gamut of problems.

We believe that such confusion has hampered the curious nature of researchers to explore

cloud computing. With a vague assumption that there aren't any challenges that have not been previously posed by various distributed computing platforms such as compute-clusters and grid computing, many of the HPC researchers have not been motivated enough to explore the newer research challenges and opportunities in offering computing as a utility. Furthermore, the lack of universal development standards for cloud computing platforms mandate the eScience developers to rewrite their respective applications from scratch for every cloud offering.

There have been some initial work to understand the pros and cons of this new framework [4, 11, 12] of cloud computing. Although cloud computing has many aspects closely similar to traditional parallel and distributed computing platforms, it poses a new set of its own challenges. Traditional large-scale computing resources were not targeted at enabling end-users to rent compute hours with provisioning time being in minutes. On the contrary, cloud computing facilitates anyone to experiment with an idea on a massive platform without investing the capital in owning the resources first, thereby it has the potential to target a much bigger set of users not necessarily familiar with the parallel or distributed computing aspects. Moreover, in order to enable the HPC researchers who currently work with large distributed computing systems, but do not work with cloud computing, to bring their expertise to cloud computing it is essential to provide them with easier means of applying their knowledge to cloud computing. Therefore, we propose to create the frameworks and resources where eScience developers and researchers can effectively build scientific applications without having the in-depth knowledge of a platform's internal architecture and APIs. These applications will also support portability from one cloud vendor to another with little to no change in the code.

The primary goals of this Ph.D. thesis are

1. creating a generic framework for bag-of-tasks HPC applications to serve as the basic building block for application development on the Azure cloud platform,

2. creating a set of APIs for HPC application development over cloud platforms, which is similar to message passing interface (MPI) from traditional parallel and distributed

setting, and

3. implementing *Crayons* as the first ever end-to-end parallel scientific application to parallelize the fundamental GIS operations of map overlay as a proof-of-concept to show the efficacy of the frameworks and the APIs created as part of this thesis.

PART 2

STATE-OF-THE-ART

Although there is no consensus on the definition of cloud computing [13], it is typically perceived as a set of shared and scalable commodity computing resources that are geographically located throughout the world and are available on-demand over the web [3]. According to NIST [6] a cloud computing platform should have five characteristics (1) on-demand self-service for customers as needed automatically, (2) access to the network over a broad range of clients such as mobile phones, laptops and PCs, and PDAs, (3) resource pooling for different types of computing resources, (4) rapid elasticity, to support rapid provisioning and release of capabilities, and (5) Measured service to support transparent utility based (pay-as-you-go) service.

2.1 Cloud Computing

As discussed previously, the term cloud computing started getting popular around the year 2007 and since then its popularity has only increased. Figure 1.1 illustrates the web search interest growth rate of cloud computing (blue line) as compared to virtualization technology (red line) or grid computing (yellow line) according to Google search Insights [14]. As reported in [7] cloud computing outperformed grid computing in terms of growth rate of web search interest in mid 2008. Over the next few months cloud computing also encapsulated virtualization and hence a decrease in the growth rate of web search interest for virtualization was seen.

Since cloud computing offers researchers to think of research questions without worrying about the scale of resources required, it has drawn wide interest from researchers, especially those working with data and compute-intensive scientific applications [8, 15, 16]. The resources are made available to end users as various services - ‘‘Platform as a service’’

(PaaS), ‘‘Infrastructure as a service’’ (IaaS), and ‘‘Software as a servic’’ (SaaS) etc. Conceptually, the key idea here is to abstract the provisioning mechanism at a level where users can avail these resources as and when needed without worrying about the scale, availability, or maintenance. Most often, users are billed for utilization based on the time a resource was reserved by a user.

Srirama et. al. [17] have designed a framework to transform the existing resources at universities into private clouds to empower students and researchers to efficiently utilize these resources. Not only academia but commercial vendors have also recognized the pervasiveness of cloud computing in near future, many of the vendors have rolled out their cloud computing based services. Rimal et. al., [9] have done a comprehensive comparison of various cloud computing vendors including Amazon EC2, Microsoft Azure, Google App Engine, IBM Blue Cloud, Nimbus, 3Tera, and Gigaspaces among others. Gaming industry has also shown interest in porting games to cloud so that games can be streamed over the web [18]. With the computationally intensive aspects of games moved to cloud, users can play high performance games that require expensive high end computing resources over relatively cheaper computers and other networked devices.

A number of scientific applications have also been ported to cloud, but yet it is far from what has been achieved using traditional large compute-clusters. Rehr et. al [16] studied the feasibility of porting two scientific applications - x-ray spectroscopy and electronic structure code FEFF - on Amazon EC2. Hoffa et. al. [19] have reported the advantages and shortcomings of using cloud computing for an application called Montage from the field of astronomy and compared the performance of Montage over local and virtual cluster.

Researchers have identified the problem with lack of standardized tools, access mechanisms, and runtime environments to work with various cloud platforms. Some of the research projects have worked on creating generic APIs that create an abtraction level and connects these APIs to the APIs of the respective cloud platform. For instance, projects such as Apache Whirr[20] and jClouds[21] offer cloud-neutral APIs that allow users to write portable codes using their APIs. However, the biggest challenge for adoption of cloud com-

puting is the initial learning curve to get started with cloud computing, which is not solved by these APIs. Since the users still need to understand the unique architecture and at least one set of APIs, these projects are useful for developers who are familiar with at least one cloud platform and now either want to move on to another cloud or want to write portable applications.

2.2 Programming models for the cloud platforms

There have been many programming models inspired by the general principle of message passing involving point-to-point and/or collective communication such as Chimp[22] and PVM[23] and MPI[24]. However, after the adoption of MPI standard it became the de-facto standard for parallel programming[25]. There have been a large number of MPI implementations by various organizations[26–29]. The primary reason for such popularity of MPI standard was the ease of use and the simplicity of design.

Unfortunately, there is a dearth of programming models tailored for the cloud platforms. While map-reduce framework [30] has shown good potential to be the platform of choice for cloud platforms, it is not natively supported by many popular cloud platforms.

Similarly, the MPI framework has faced the performance challenge as the commodity hardware based architecture of cloud platforms is not a good fit for traditional MPI implementations. As a matter of fact, the cloud computing paradigm is far from the adoption that MPI implementations have received over traditional platforms. We believe the primary reason is the steep learning curve due to lack of uniformity among various cloud platforms as well as the unique underlying architecture. Moreover, the lack of familiar tools and runtime environments make it difficult for an application developer to write an application for the cloud platforms.

Furthermore, the traditional implementations of MPI frameworks are not suitable for the cloud platforms. The high latency and low bandwidth network of the cloud architecture does not work well with MPI and other message passing models that rely on the efficiency of the network. Authors in [31] demonstrate that the cloud platforms such as Amazon EC2

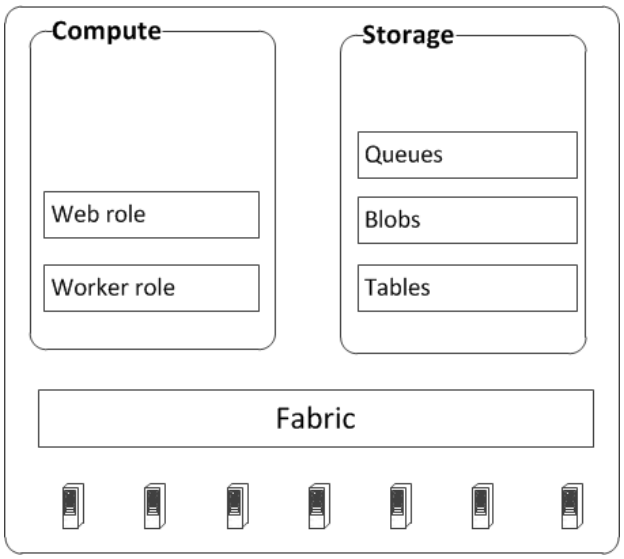


Figure 2.1 Typical programming artifacts of Windows Azure platform [1]

could be as much as 35X slower than the traditional platforms when it comes to network bandwidth and latency.

2.3 Windows Azure cloud platform

The Windows Azure cloud platform is a computing and service platform hosted in Microsoft data centers. Its programming primitives consist of two types of processes called *web role* and *worker role* for computation, a variety of storage mechanisms, and the Windows Azure Fabric. The typical programming artifacts of Windows Azure platform are shown in Figure 2.1. The web role acts as a web application accessible over HTTP and HTTPs endpoints and usually is the front end of any Azure cloud based application. Worker roles are the processing entities representing the backend processing for the web application. A web role is hosted in an environment with support for a subset of ASP.NET and Windows Communication Foundation (WCF) technologies[1]. Both web role and worker role processes can have different configurations as shown in Table **??**.

Fabric (Windows Azure Fabric) in the Windows Azure platform is the network of interconnected physical computing nodes consisting of servers, high-speed connections, and switches. Compute and storage components are part of the Fabric.

Table 2.1 Listing of virtual machine configurations available for web role and worker role instances with Windows Azure

VM Size	CPU Cores	Memory	Storage
Extra Small	Shared	768MB	20 GB
Small	1	1.75 GB	225 GB
Medium	2	3.5 GB	490 GB
Large	4	7 GB	1000 GB
Extra Large	8	14 GB	2040 GB

The storage objects are organized as services and can be accessed by both web roles and worker roles. There are three primary types of storage services: Queues, Blobs, and Tables. Additionally, local storage can be configured for role instances. Azure platform also provides a caching service to temporarily hold data in memory across different servers. In this paper, we only concentrate on the three primary storage services.

Queues are similar to the traditional queue data structure, but first-in first-out (FIFO) functionality is not always guaranteed. Queues are prominently used for communication among instances of web roles and worker roles. A reference to a task is usually put as a message on the queue and a set of worker role instances are deployed to process them. The blob storage is a persistent storage service like a traditional file; the data can be stored as a collection of small blocks of size up to 4 MB or large pages of size up to 1 TB. Azure tables arrange data into a structured organization and thus are useful for query-based data management.

PART 3

AZUREBENCH: BENCHMARKING SUITE FOR THE AZURE CLOUD PLATFORM

The consistent growth of Cloud computing hints that it is poised to be the platform of choice for future generations of High Performance Computing (HPC) application development. A large number of vendors have rolled out their commercial cloud infrastructures. However, the service offerings are usually only best-effort based, without any performance guarantees. Unlike traditional computing platforms where the user had access to both hardware and software, cloud computing only provides resources as a service and hence the quality of service, performance consistency, and resource sharing is soleley controlled by the cloud vendor.

Since developers do not control the behavior of other tenants on the machine where their applications are hosted, the verifiability of the services promised becomes a crucial question to choose cloud computing over traditional computing platforms. Cloud computing effectively saves the eScience developer the hassles of resource provisioning but utilization of these resources will be questionable if it can not meet the performance expectations of deployed applications. Furthermore, in order to make application design choices for a particular cloud offering, an eScience developer needs to understand the performance capabilities of the underlying cloud platform. Among all clouds, the emerging Azure cloud from Microsoft remains a challenge for HPC program development both due to lack of its support for traditional parallel programming support such as MPI and map-reduce and due to its evolving APIs. To aid the HPC developers, we present an open-source benchmark suite, AzureBench[1], for Windows Azure cloud platform. We report comprehensive performance analysis of Azure cloud platform's storage services which are its primary artifacts for inter-

[1]This work is partially supported by NSF CCF 1048200 and Microsoft.

processor coordination and communication. We also report on how much scalability Azure platform affords using up to 100 processors and point out various bottlenecks in parallel access of storage services. The chapter also has pointers to overcome the steep learning curve for HPC application development over Azure. We also provide an open-source generic application framework that can be a starting point for application development for bag-of-task applications over Azure.

There are several vendors offering cloud services in the market today. The service offerings differ in terms of software support, platform support, and also developmental tools support. While some vendors empower the developer to deploy his/her own virtual machine images, some others provide their own APIs to interact with their cloud services. Moreover, some vendors support traditional programming support such as MPI and Map-reduce while others have different software infrastructures. Therefore, it is critical for an eScience developer to choose the cloud platform that is most suitable for his/her application based on the availability of resources required for development and maintenance of that application.

In this chapter we present AzureBench[2] - a suite of benchmarks for Azure platform's storage services. We provide a comprehensive scalability assessment of Azure platform's storage services using up to 100 processors. Our work extends a preliminary study by Hill et al. [32] in 2010 and provides additional key insights. Our assessments of Azure platform provide updated, realistic performance measurements as we utilize the APIs released after significant changes were made to the Azure cloud platform since 2010. Some of the earlier restrictions of Azure platform's storage services, such as expiration of a message in *Queue* storage after 2 hours, rendered Azure platform problematic for long-running real-world scientific applications.

AzureBench is an open-source benchmark suite hosted at Codeplex repository[3] available under GPLv2. The open-source nature will motivate further research in this direction. We have chosen not to include the assessment of operating cost and SQL-Azure functionalities

[2]Please visit the project page at `http://www.cs.gsu.edu/dimos/content/azurebench.html` for up-to-date information including code, data, and plots.

[3]Source code for AzureBench can be downloaded from `http://azurebench.codeplex.com/`.

in this study. Moreover, comparison with other cloud platforms is also not studied in this work - primarily due to the differences in architectures. We plan to address both these issues by including it in AzureBench and provide a detailed report in near future.

The Windows Azure cloud platform provides three types of storage services: Blob storage, Queue storage, and Table storage. We have done an extensive study of all three storage mechanisms for varying load and compute instances.

In order to evaluate Windows Azure storage mechanisms, we deploy varying number of virtual machines (VM) and these virtual machines read/write from/to Azure storage concurrently. For the sake of fair comparison, we maintain the same amount of input and output load throughout the benchmarking process.

Azure platform performs impressively better than what has been reported earlier [32], as many of the previous drawbacks have been addressed by Microsoft. Nevertheless, there are still few bottlenecks left in Azure storage mechanisms that we have discovered. We also lay out a simplistic generic application framework for Azure cloud to give developers a starting point to design their own applications. Finally, we provide a summary of our findings and make recommendations for developers to efficiently leverage the maximum throughput of storage services.

We do not analyze the local storage feature at each compute resource as it is similar to writing to the local hard disk and thus does not add value to the study. Since we are only interested in assessment of Azure storage in this study, the compute instances do not perform any computationally-intensive task on the data. The data generation is limited to minimum possible and the time spent in data generation is also ignored. We have done another study to analyze the performance of Azure platform for both computational and I/O phases with respect to a scientific application from Geographic Information System and Science (GIS) domain. Motivated readers can refer to [33] to read more details about that study.

The rest of this chapter is organized as follows: Section 3.1 discusses the previous work reported in literature. We present a generic framework for HPC applications on Windows Azure cloud platform in Section 3.2. Our experimental results and analysis is presented is

Section 3.3. Section 3.4 concludes this chapter with comments on the future work.

3.1 Background and Literature

3.1.1 Cloud Computing for Scientific Applications

Cloud Computing is typically perceived as a set of shared and scalable commodity computing resources, located all over the world and available on-demand over a network. The resources are made available to end users as various services such as "Platform as a service" (PaaS), "Infrastructure as a service" (IaaS), and "Software as a service" (SaaS). Conceptually, the key idea here is to abstract the provisioning mechanism at a level, where users can avail of these resources dynamically without burdening themselves with either the availability or the maintenance. There is yet no universally accepted definition of Cloud computing [3].

The term cloud computing started getting popular around the year 2007, its popularity has only increased since then. Figure 1.1 illustrates the web search interest growth rate of cloud computing (blue line) as compared to virtualization technology (red line) or grid computing (yellow line), according to Google search Insights [14]. Cloud computing outperformed grid computing in terms of growth rate of web search interest in mid 2008. Over the next few months, cloud computing also encapsulated virtualization and hence a decrease in the growth rate of web search interest for virtualization was seen.

Thanks to the dynamic provisioning of resources, cloud computing has drawn wide interest from researchers, especially those working with data and compute-intensive scientific applications [8, 15, 16]. Users are billed for utilization, largely based on the time a resource was reserved by a user.

Srirama et al.[17] report how universities can utilize the existing HPC resources as their own private clouds. Ekanayake et al. [34] have created a framework for iterative map-reduce on Azure and have demonstrated, by way of samples, how their framework can be utilized to port map-reduce based applications to Azure.

Commercial vendors, similarly, have also recognized the importance of cloud computing; many of the vendors have already rolled out their cloud computing based services. Rimal et al. [9] have done a comprehensive comparison of various cloud computing vendors including Amazon EC2, Microsoft Azure, Google App Engine, IBM Blue Cloud, Nimbus, 3Tera, and Gigaspaces among others. Gaming industry has also used cloud platform to host compute intensive aspects of games to enable a rich gaming experience without the need of expensive computing resources [18].

A number of HPC scientific applications have also been ported to cloud. Rehr et al.[16] studied the feasibility of porting two scientific applications - X-ray spectroscopy and electronic structure code FEFF - on Amazon EC2. Hoffa et al.[19] have reported the advantages and shortcomings of using cloud computing for an application called Montage from the field of astronomy and compared the performance of Montage over local and virtual cluster.

3.2 An Application Framework for Scientific Applications on Windows Azure Cloud Platform

Figure 3.1 shows a generic framework for application development on Windows Azure cloud platform. Application workflow for Azure cloud based applications typically starts with a web-interface where users have an option to specify the parameters for background processing. Moreover, this interface should be interactive to update users with current state of the system, especially for time consuming applications. This is typically achieved by employing a web-role, although some applications use command line interface where this component could be missing from the application framework.

VM configuration for web role depends on the intensity of the tasks to be handled by the web role. For applications where web role performs computationally-intensive operations, a fat VM configuration should be chosen. Similarly, if the web role needs to access large data items from cloud storage, it could be a fat VM to upload/download data to/from the storage using multiple threads.

To communicate task with worker roles, web role puts a message on a *Task assignment*

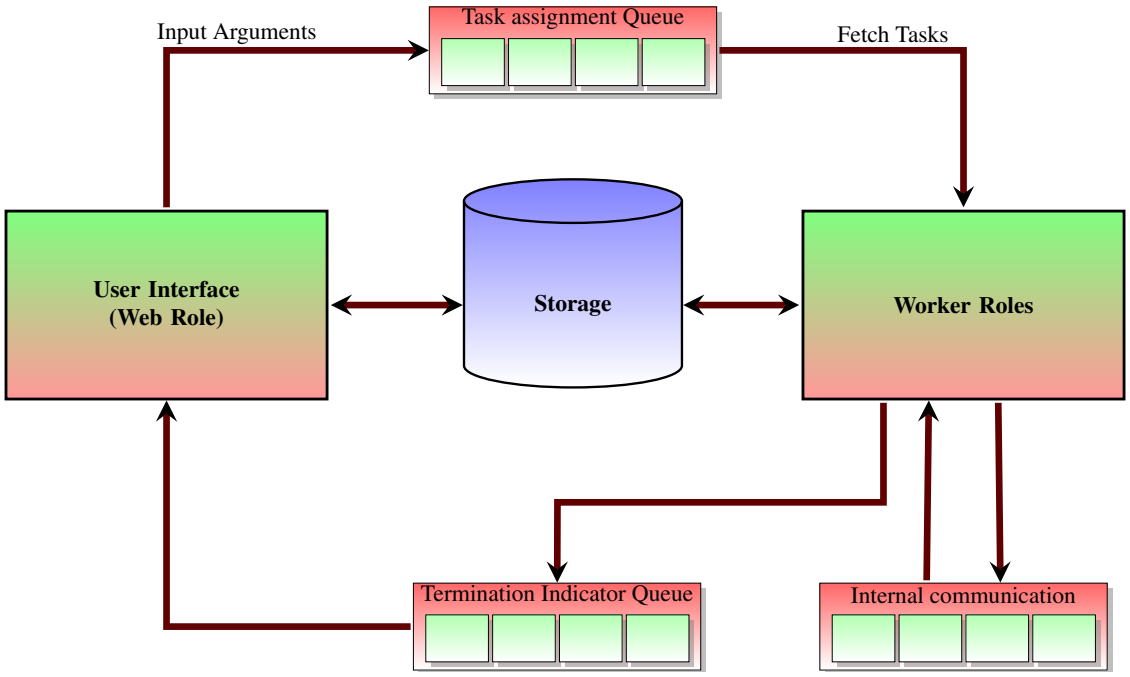


Figure 3.1 A Generic Application Framework for Scientific Applications on Windows Azure Cloud Platform

queue as shown in Figure 3.1. If there are distinct input parameter sets, there could be multiple task assignment queues for each set of parameters.

Worker role instances keep checking this queue and as soon as they locate a message there, they start background processing based on the content of the message in the queue. Worker roles communicate with the storage services to acquire the data required for processing.

One or more queues can be utilized to communicate among worker role instances. Since one role instance cannot automatically query the state of other role instances in Windows Azure, the communication depends on storage services - typically Queue storage. Azure platform also supports TCP endpoints that can be configured to facilitate an application to listen on an assigned TCP port for incoming requests. TCP messages can be sent/received among Azure roles or can be used for communication with external services - these messages are not currently studied in this work.

For a time-consuming interactive application, it is essential to update the user interface.

To achieve this, a worker role instance can put a message on a queue after every phase of processing completes. The web role can read the number of messages in this queue and accordingly update the user interface. This queue is shown as *Termination Indicator Queue* in Figure 3.1.

The effectiveness of this framework has been proven in several applications, such as our own GIS application Crayons [33] and Twister4Azure[34].

3.3 Benchmark Experiments and Timing Characteristics

Windows Azure storage services partition the stored data across several servers to provide enhanced scalability. The absolute limit on a storage account is 100 TB. However, there are additional limits on scalability targets. Windows Azure storage services can handle up to 5,000 transactions (entities/messages/blobs) per second. Moreover, there is a maximum bandwidth support for up to 3 GB per second for a single storage account. Exceeding any of the specified limits result in the failure of a role instance.

In this section, we detail our analysis with the performance test of all three Azure storage services: Blob storage, Table storage, and Queue storage.

3.3.1 Blob Storage

Blob storage in Windows Azure is similar to the traditional file system. Blob storage service, organized into a hierarchy, can be used to store large amount of unstructured data. One storage account can have multiple blob containers, and one container can store multiple blobs. Blobs are partitioned based on "container name + blob name" combination, i.e, each individual blob can be stored at a different server for maximum throughput. The throughput of a blob is up to 60 MB per second.

There are two types of blobs in Windows Azure: Block blobs and Page blobs. Block blobs can be created in two ways - Block blobs less than 64 MB in size can be directly uploaded to blob storage as a single entity, and Block blobs greater than 64 MB can be uploaded as a set of multiple blocks of size up to 4 MB each. There can be a total of 50,000

Algorithm 1 Azurebench blob benchmarks

$syncCount := 0$
for $repeat := 1 \rightarrow 10$ **do**
  $BlockBlob :=$ ''AzureBenchBlockBlob''
  $PageBlob :=$ ''AzureBenchPageBlob''
  Total blocks/pages in a blob $count := \left(\frac{100MB}{1MB}\right)$
  /* Page blob upload */
  $content := randomData(1MB)$
  for $pageid := 1 \rightarrow \left(\frac{count}{worker}\right)$ **do**
    PutPage$(PageBlob,\ content)$
  end for
  /* Block Blob Upload */
  $content := randomData(1MB)$
  for $blockid := 1 \rightarrow \left(\frac{count}{worker}\right)$ **do**
    PutBlock$(BlockBlob,\ content)$
    Add $blockid$ to $blockIdList$
  end for
  PutBlockList$(blockIdList)$
  $Synchronize(++ syncCount)$
  /* Downloading pages from a Page blob randomly */
  for $pageID := 1 \rightarrow count$ **do**
    $pageOffset := randomNumber(1,\ count)$
    $Page :=$ ***GetPage***$(PageBlob,\ pageOffset)$
  end for
  /* Downloading blocks from a Block blob */
  for $blockID := 1 \rightarrow count$ **do**
    $Block :=$ ***GetBlock***$(BlockBlob,\ blockID)$
  end for
  $Synchronize(++ syncCount)$
  /* Download entire Page Blob */
  Download $PageBlob$ using ***PageBlob.openRead()***
  /* Download entire Block blob */
  Download blob using ***BlockBlob.DownloadText()***
  $Synchronize(++ syncCount)$
  DeletePageBlob$(PageBlob)$
  DeleteBlockBlob$(BlockBlob)$
end for

such blocks in a blob. Thus, the maximum size of a Block blob cannot exceed 200 GB.

The Page blob artifact was not there in the Blob storage initially; it was later introduced to facilitate random read/write operations on blobs. A Page blob is created and initialized with a maximum size; pages can be added at any location in the blob by specifying the offset. The offset boundary should be divisible by 512, and the total data that can be updated in one operation is 4 MB. A Page blob can store up to 1 TB of data. We perform both, upload and download, tests on both types of blobs.

Algorithm 1 shows the skeleton of our benchmark code for Azure Blob storage (Azure APIs highlighted in bold itaiics). Each worker role starts with uploading one 100 MB blob to cloud storage in 100 chunks (blocks or pages) of 1 MB each.

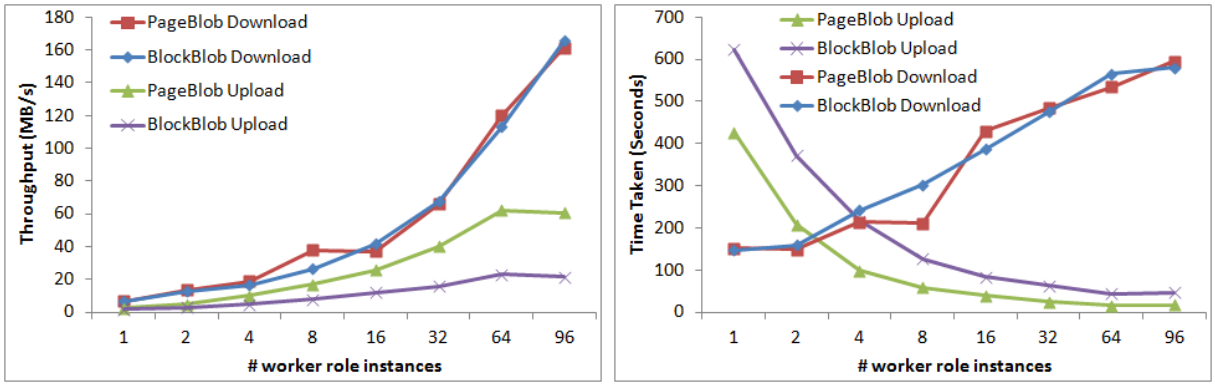
To ensure that the process of downloading starts only after the process of uploading has finished, worker roles need to synchronize. Synchronizing among worker roles in Azure platform is an interesting process by itself. There is no API in the Azure software development kit that provides a traditional barrier like functionality. However, a queue can be used as a shared memory resource to implement explicit synchronization among multiple worker role instances. Each worker can put a message on a designated queue that acts as a barrier. When the number of messages in the queue is equal to the number of workers, it is safe to assume that all workers have touched the barrier and hence all of them can cross it.

However, what makes it interesting is that if the workers delete the messages after exiting the *While* loop, those workers that have put the message in the queue, but yet to exit the loop, will never meet the loop termination condition. On the other hand, if the workers do not delete the messages, the number of workers will never match the number of messages in the queue after first synchronization cycle. Therefore, in our case, for each synchronization phase, we pass a variable that accounts for the messages left in the queue during previous synchronization phases. Moreover, since a large number of requests to get the message count can throttle the queue, each worker sleeps for a second before issuing the next request. The time reported in the experiments does not include the time spent in synchronization. Our synchronization mechanism is illustrated in Algorithm 2.

Algorithm 2 Synchronization among worker role instances

Input: *syncCount*

```
syncQueue :="Termination_Indicator_Queue"
arrived := 0
while arrived < (workers * syncCount) do
  arrived := GetMsgCount(syncQueue)
  Sleep(1 second)
end while
```



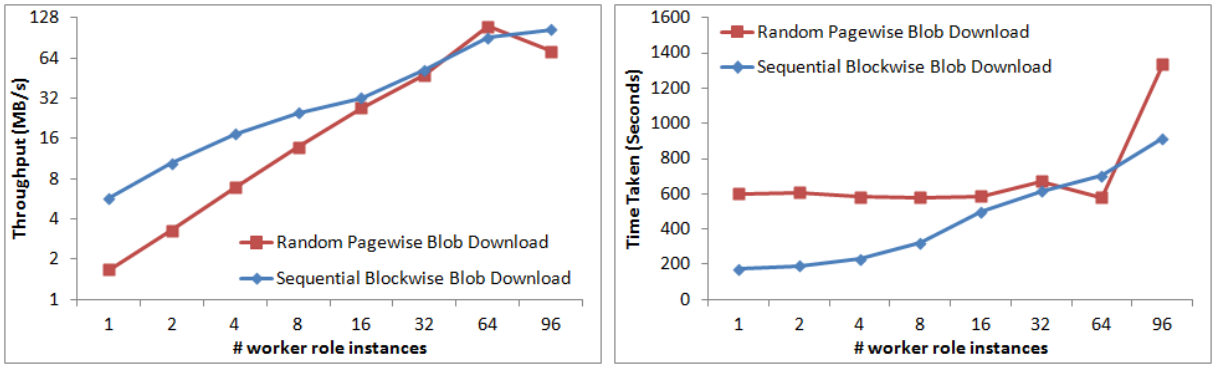
(a) Throughput (b) Time

Figure 3.2 Azurebench Blob storage benchmarks

Figure 3.2 shows the performance analysis of Azure platform's Blob storage service. The total uploaded data to the Blob storage is 2 GB - 1 GB for each Block and Page blob. The downloaded data, however, is 2 GB per worker role instance. Since each worker downloads the blobs from the Blob storage, the download time increases with increasing number of worker role instances for both Block and Page blobs as shown in Figure 3.2(b). However, the throughput of the Blob storage also increases with increasing number of worker role instances, as shown in Figure 3.2(a).

The upload time reduces with the increasing number of workers, as the amount of the data to be uploaded per worker reduces. Moreover, the increasing throughput for the process of uploading suggests that the Blob storage scales well even when multiple clients are trying to upload data.

The maximum throughput for blob download process was 165 MB/s, achieved for Block



(a) Throughput (b) Time

Figure 3.3 Blob download using one page/block at a time

blob download using 96 workers, and the maximum throughput for blob upload process was 60 MB/s, realized for Page upload process using 96 workers. The maximum throughput for a Block blob upload process was only a little over 21 MB/s using 96 workers; the reason why Page blob upload process demonstrates superior upload throughput is the capability of Page blobs to allow fast random access to read/write pages.

To evaluate the performance of random access for Page blob download process, each worker downloads 100 random pages from the Page blob that was uploaded previously. Since Block blobs do not support random access of blocks, we let each worker read one block at a time sequentially.

Figure 3.3 shows the download time and throughput of Blob storage when blobs are downloaded by accessing one block/page at a time. The pages from the Page blob are accessed randomly, which adds the overhead of locating the page in a Page blob. The blocks from the Block blob are accessed sequentially. The maximum throughput achieved by Page wise blob downloading was more than 71 MB/s using 96 workers. The Block wise blob downloading for the same amount of worker roles was more than 104 MB/s.

3.3.2 Queue Storage

In Windows Azure platform based applications, queues are used by both web role and worker roles to communicate with each other or among different instances of the same role. The distinguishing feature of an Azure Queue storage from the traditional queue data structure is its lack of ability to guarantee a FIFO operation. This lack of guarantee for FIFO operation can cause issues if a queue is to be used to signal a special event. For instance, if web role wants to put a message at the end of the task queue to signal the end of work, it might not work as expected. Since FIFO is not guaranteed, the worker roles might read this message before the actual messages for tasks and hence quit processing while there is work in the task pool. To achieve termination signaling, it is recommended to create a dedicated termination indicator queue where worker instances can send messages to signal an event.

A storage account can have unlimited number of uniquely named queues. Each queue can have unlimited number of messages (limited by 100 TB limit of a storage account) and each message has a visibility timeout period. Queues are partitioned based on queue names, i.e. a single queue and all the messages stored in it are stored at a single server. A single queue can only handle up to 500 messages per second. Thus, if an application only interacts with queues, it is essential to employ multiple queues for better scalability.

The messages in a queue can be consumed by any service, but the consumer is expected to delete the message after processing. Once a message is read, it is made invisible from the queue for other consumers. A consumer also has the option to peek a message rather than reading it, in which case the message stays visible in the queue for other consumers. If the consumer does not delete the message after its consumption, it reappears in the queue after a certain time.

Similarly, if a message is left in the queue for longer than a week (the duration was 2 hours for previous APIs), it automatically disappears. The maximum size of a message supported by Azure cloud is 64 KB - it used to be 8 KB prior to October 2011 version of Azure APIs. To store larger data, one can store the actual data in Blob storage and put the blob's name in the queue as a message. Equipped with these properties, queues can easily

Algorithm 3 Azurebench Queue storage benchmarks with a separate queue per worker

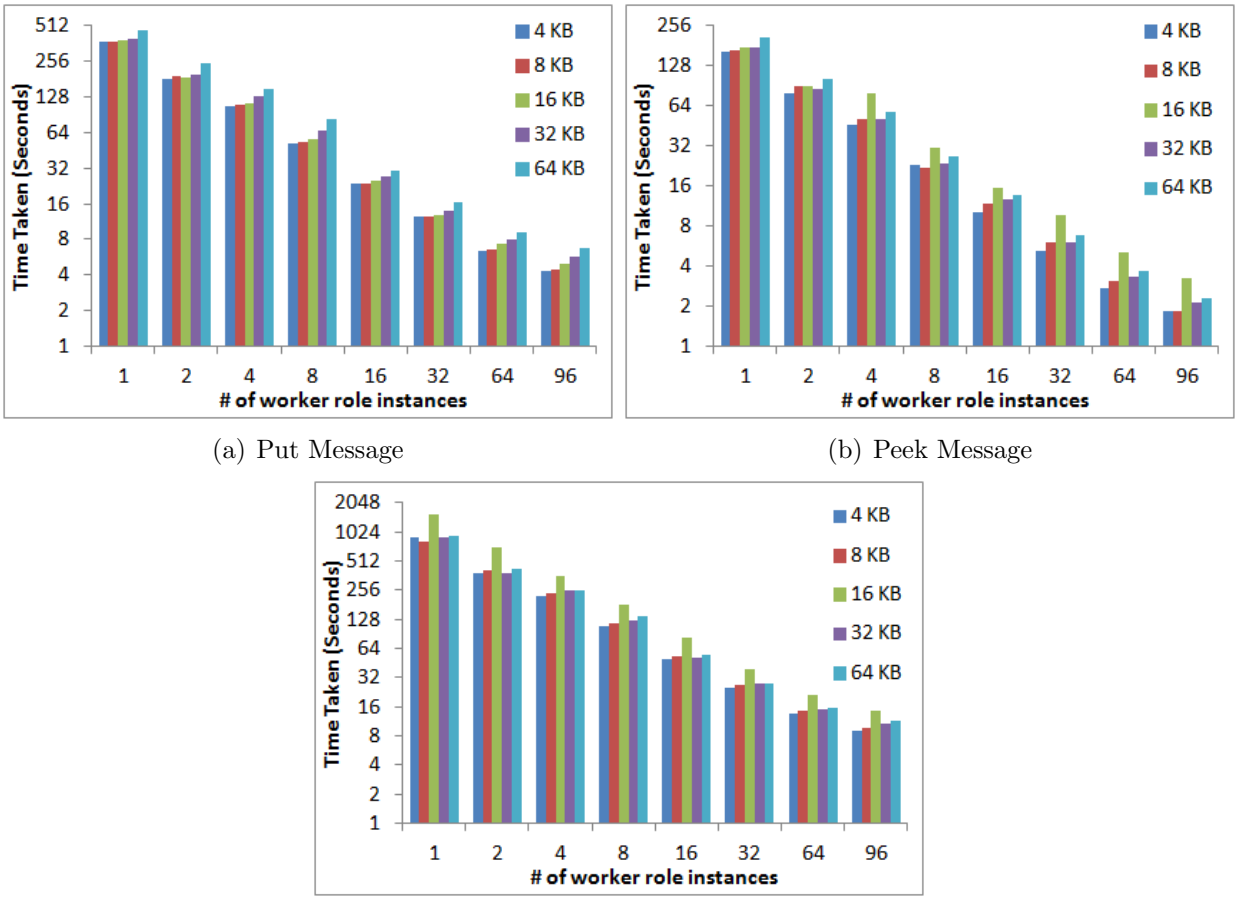
```
QueueName := "AzureBenchQueue + roleID"
Message_Size := 4KB
Message_Count := (20,000 / workers)
CreateQueue(QueueName)
for repeat := 0 → (log(64KB / 4KB) := 4) do
  for count := 1 → Message_Count do
    Message := randomData(Message_Size)
    PutMessage(QueueName, Message)
  end for
  for count := 1 → Message_Count do
    Message := PeekMessage(QueueName)
  end for
  for count := 1 → Message_Count do
    Message := GetMessage(QueueName)
    DeleteMessage(Message)
  end for
  Message_Size := Message_Size * 2
end for
DeleteQueue(QueueName)
```

facilitate the behavior of a shared task pool with in-built fault tolerance mechanisms.

For our experiments, we test three operations on Azure queues: inserting a message using *PutMessage* API, reading a message using *GetMessage* API, and reading a message using *PeekMessage* API. Concurrent consumers can read a message from a queue using *PeekMessage* API. However, if a consumer reads a message using *GetMessage* API, only first consumer can read the message as the message becomes invisible from the queue for a certain amount of time, defined at the time of message creation.

We have evaluated Queue storage under two scenarios, (i) each worker works with its own dedicated queue, and (ii) all workers access the same queue. For both experiments, a total number of 20K messages were first inserted in the queue, then read using both APIs, and finally deleted from the queue.

Algorithm 3 represents the first scenario - each worker has its own queue. In this experiment, we evaluate the performance of the queue storage with varying size of messages - 4 KB, 8KB, 16 KB, 32 KB, and 64 KB. Interestingly, 48 KB (49152 Bytes to be precise) is



(a) Put Message

(b) Peek Message

(c) Get Message

Figure 3.4 Azurebench queue benchmarks - Separate queue per worker

the maximum usable size of an Azure queue message, rest of the message content is metadata. The total real data uploaded (and downloaded) to (from) Queue storage in an experiment with message size of 48 KB is around 1.2 GB.

Figure 3.4 shows how Windows Azure platform's Queue storage scales with varying load and varying number of worker role instances. Impressively, the Queue storage scales very well for varying message sizes (4 KB through 64 KB) and varying number of worker role instances for all three operations - insertion, reading using *PeekMessage*, and reading using *GetMessage*. We also tried the same experiment with 200 messages in total and the results were still the same. The experiments at different times also demonstrated similar behavior.

Windows Azure platform maintains three replicas of each storage object with strong consistency[35]. Figure 3.4(a) shows the time to put a message on the queue. For *Put Message* operation, the queue needs to be synchronized among replicated copies across different servers. Figure 3.4(b) shows the behavior of *Peek message* operation. It is the fastest of all three operations, as there is no synchronization needed on the server end. The *Get Message* operation, as shown in Figure 3.4(c), is the most expensive operation as in this case, in addition to synchronization, the message also becomes invisible from the queue for all other worker role instances, and hence extra state needs to be maintained across all copies. Moreover, in our case the *Get Message* operation also includes deletion of the respective message.

One interesting case is of message size 16 KB. Surprisingly, the *Get* operation for this sized messages took significantly more time than other message sizes (both smaller and larger ones). We do not know the reason behind this, but this was consistently seen in all repeated experiments.

Algorithm 4 illustrates the steps of evaluation where multiple workers concurrently interact with a single queue. Each worker accesses the queue once and then spends a certain amount of time before going back to the queue again. This behavior simulates a real world application, where the application accesses the queue intermittently during the course of execution. We varied the time taken by a worker before going back to the queue from 1 second to 5 seconds; the reported time only includes the time spent in communication with the queue. We ensured that the total number of transactions remain same irrespective of number of workers. Thus, workers proportionately carried out fewer transactions on the shared queue as their number increased. The message size for this experiment was kept constant at 32 KB. Additionally, in order to ensure that the number of transactions between the workers and the queue never exceed the bandwidth limit of 500 messages per second, each operation is split into multiple rounds.

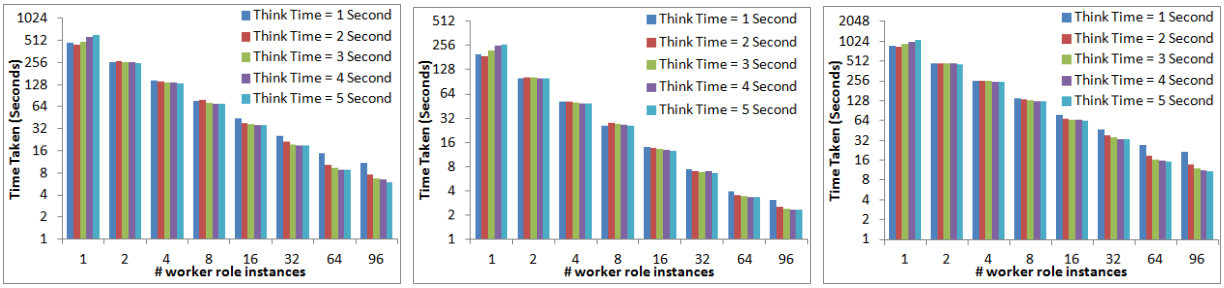
Figure 3.5 shows the behavior of Queue storage when multiple workers are accessing a queue in parallel. Parallel access of a queue increases the contention at the queue, hence the

Algorithm 4 Azurebench queue benchmarks with a single queue shared among multiple workers

$QueueName$:="AzureBenchQueue"
$Message\_Size := 32KB$
$Message\_Count := \left(\frac{500}{workers}\right)$
$rounds := \left(\frac{20{,}000}{500}\right)$
$thinkTime := 1\ second$
for $repeat := 1 \rightarrow \left(\left(\frac{5\ seconds}{1\ second}\right) := 5\right)$ **do**
  for $round := 1 \rightarrow rounds$ **do**
    for $count := 1 \rightarrow Message\_Count$ **do**
      $Message := randomData(Message\_Size)$
      PutMessage$(QueueName, Message)$
    end for
    $think(thinkTime)$
    for $count := 1 \rightarrow Message\_Count$ **do**
      $Message :=$ ***PeekMessage***$(QueueName)$
    end for
    $think(thinkTime)$
    for $count := 1 \rightarrow Message\_Count$ **do**
      $Message :=$ ***GetMessage***$(QueueName)$
      DeleteMessage$(Message)$
    end for
    $think(thinkTime)$
  end for
  $thinkTime := thinkTime + 1\ second$
end for

time taken by each operation is greater than the time taken when each worker accesses its own queue (Figure 3.4). The think time also plays a vital role in realizing the performance of a queue. It can be seen from Figure 3.5, that the time taken by an operation reduces as the think time increases; in some cases, the time reduces by a factor of almost two. As the number of workers starts increasing, the time starts decreasing. This is not due to any reduction in overall parallel access to the shared queue. This demonstrates that the queue implementation scales very well at these access frequencies.

The first scenario, where we use separate queues for each worker role instance, each worker can put a parallel request as the queues are partitioned based on queue names. This is the reason why we see super-linear speedup in many cases. Consequently, we recommend



(a) Put Message (b) Peek Message (c) Get Message

Figure 3.5 Azurebench queue benchmarks - Single shared queue

usage of multiple queues as and when possible to make efficient use of Queue storage.

3.3.3 Table Storage

Table storage provides semi-structured data storage in Azure cloud platform. A table is comprised of entities of up to 1 MB in size; each entity is composed of up to 255 properties. A table can be queried based on the default properties - *Row Key* and *Partition Key* - which apparently also form the unique key for an entity. Unlike traditional database tables, Azure Table storage does not have a schema. All of the properties of a table are stored as *(Name, Value)* pairs, i.e. two entities in the same table can have different properties.

Tables are partitioned on the partition keys, i.e. entities of a table that belong to the same partition are stored together on a server. A single partition can support access to a maximum of 500 entities per second. Therefore, a good partitioning of a table can significantly boost the performance of Table storage.

Algorithm 5 shows the structure of our benchmark tests for Table storage. Each worker role instance inserts 500 entities in the table, all of which are stored in a separate partition in the same table. Once the insertion completes, the worker role queries the same entities 500 times. After the querying phase ends, the worker role updates all of the 500 entities with newer data. Finally, all of these entities are deleted. The exact experiment is repeated 5 times with varying entity sizes. We have experimented with entity sizes of 4 KB, 8 KB,

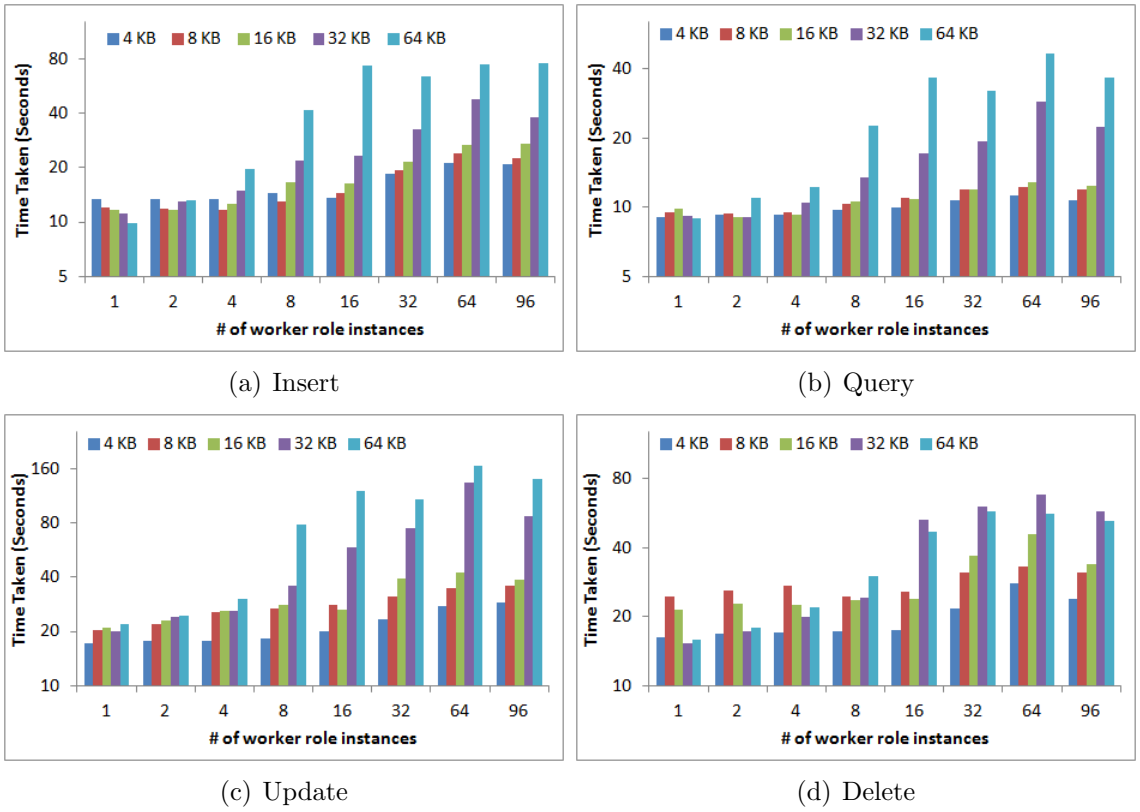
Algorithm 5 Azurebench table benchmarks

```
TableName = ''AzureBenchTable''
Entity_Size = 4KB
Entity_Count = 500
for repeat := 1 → 5 do
  for rowKey := 1 → Entity_Count do
    Entity = randomData(Entity_Size)
    Entity.partitionKey = roleID
    AddRow(TableName, Entity, rowKey)
  end for
  for rowKey := 1 → Entity_Count do
    Entity =
      Query(TableName, roleID, rowKey)
  end for
  for rowKey := 1 → Entity_Count do
    newData = randomData(Entity_Size)
    Update(TableName,
      roleID, rowKey, newData)
  end for
  for rowKey := 1 → Entity_Count do
    Delete(TableName, roleID, rowKey)
    Save(TableName)
  end for
  Entity_Size = Entity_Size * 2
end for
```

16 KB, 32 KB, and 64 KB.

Figure 3.6 shows the performance of Table storage service. The timings are almost constant till 4 concurrent clients for all entity sizes across all four operations. It can be seen from Figure 3.6(c) that updating a table is the most time consuming process. We only tested the *unconditional updates* by using the wild card character * for *ETag* during update queries. The least expensive process is querying a table, as shown in Figure 3.6(b). For entity sizes 32 KB and 64 KB, the time taken for all of the four operations increases drastically with increasing number of worker role instances.

We initially started with inserting about 1000 entities each and experienced a small number of *server busy* exceptions during the experiments, which is an indication of hitting the 500 transactions per second limit. Therefore, we tried with only 500 transactions and



(a) Insert

(b) Query

(c) Update

(d) Delete

Figure 3.6 Table Storage

everything worked without any exception.

Similar to the previous case of Blob and Queue storage, when we run into such exceptions, the worker sleeps for a second before retrying the same operation.

Many scientific applications require a storage service that offers query like behavior, as well as well organized data storage than a simple file system. Azure Table storage provides this functionality with an impressive throughput. However, a table can only handle entities (rows) that are at most 1 MB in size and have up to 255 properties (columns); we only had one column per row for our experiments.

The name Table storage could confuse a beginner developer to expect a SQL like functionality, where the size of the table should be controlled by the limit of the storage account size - both in terms of number of entities as well as properties. Moreover, if the fundamental

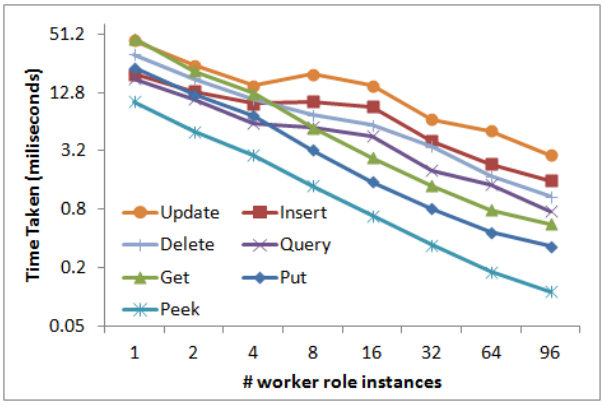


Figure 3.7 Per operation time for Table (insert, query, update, and delete) and Queue storage (put, peek, and get) services

storage unit of an application can expand beyond 1 MB, than table is not the best choice for storage - a blob should be considered. A blob can store up to 64 MB of data, or even more by arranging data into multiple blocks or pages.

Figure 3.7 shows the per operation time for Queue and Table storage services. The reported time is the average time taken by an operation, i.e. the division of total time taken by all the worker roles to finish that operation, and the number of workers. It is evident from Figure 3.7 that the Queue storage scales better than the Table storage as the number of workers increases.

3.4 Conclusion

In this chapter we have presented AzureBench - an open source benchmark suite for Windows Azure platform's storage services - along with experimental details to analyze the performance capabilities of Azure cloud platform. We have shown a comprehensive performance evaluation of Windows Azure platform's storage services - Table, Blob, and Queues. We also present a generic framework along with pointers for HPC application development on Azure.

PART 4

CRAYONS: AN AZURE CLOUD BASED PARALLEL SYSTEM FOR GIS OVERLAY OPERATIONS

Efficient end-to-end parallel/distributed processing of vector-based spatial data has been a long-standing research question in GIS community. The irregular and data intensive nature of the underlying computation has impeded the exploratory research in this space. We have created an open-architecture-based system named *Crayons* for Azure cloud platform using state-of-the-art techniques. The design and development of *Crayons* system is an engineering feat both due to (i) the emerging nature of the Azure cloud platform which lacks traditional support for parallel processing and (ii) the tedious exploration of design space for right techniques for parallelizing various workflow components including file I/O, partitioning, task creation, and load balancing. *Crayons* is an open-source system available for both download and online access, to foster academic activities. We believe *Crayons* to be the first distributed GIS system over cloud capable of end-to-end spatial overlay analysis. We demonstrate how Azure platform‘s storage, communication, and computation mechanisms can support high performance application (HPC) development. Crayons scales well for sufficiently large data sets, achieving end-to-end speedup of over 40-fold employing 100 Azure processors. For smaller, more irregular workload, it still yields over 10-fold speedup. We discuss spatio-temporal aspects, in particular employment of affinity groups for co-locating data and computation.

Since the creation of the National Spatial Data Infrastructure [36] by 1994 Presidential Executive Order 12906, the availability of digital geospatial data has increased significantly, thanks to the NSDI initiative implemented by state-wide GIS Data Clearinghouses, federal government agencies, and others. In addition to raw data, many geospatial data sets are now accessible using a variety of Open Geospatial Consortium standards [37], such as Geography

Markup Language (GML), Web Feature Service (WFS), Web Coverage Service (WCS) [38], GeoRSS and KML [39]. Figure 4.1 shows some example data sets with typical file sizes. Depending on the resolution and geographic extents, data sets can get extremely large (several terabytes) [40].

Table 4.1 Example GIS data sets and typical file sizes

Source	Example Type	Description	File Size
US Census [41]	Block Centroids	Block centroids for entire US	705 MB
	Block Polygons	2000 Block polygons for the state of Georgia	108 MB
	Blockgroup Polygons	2000 Blockgroup polygons for the state of Georgia	14 MB
GADoT [42]	Roads	Road centerlines for 5-county Atlanta metro	130 MB
USGS [43]	National Hydrography Data set	Hydrography features for entire US	13.1 GB
	National Landcover Data set	Landcover for entire US	3-28 GB
JPL [44]	Landsat TM	pan-sharpened 15m resolution	4 TB
Open Topography [45]	LIDAR	LIDAR point clouds 1-4 pts/sq. ft	0.1-1 TB

Hence, advanced tools for processing and analyzing huge dynamic spatial data are needed to take advantage of this wealth of information and to meet a wide range of end-user demands [46]. While some prior works have achieved good results on handling and modeling large raster data sets, very few address high volume vector data [47]. In addition, micro-level geospatial data are increasingly being used in simulation frameworks, where data structures and communication protocols require new approaches for the efficient and dynamic processing of potentially large and interacting data sets [48].

HAZUS-MH [49], a FEMA-developed application, is an example of a GIS software system predominantly used in US for disaster-based consequence modeling. HAZUS-MH integrates scientific and engineering disaster modeling knowledge with inventory data in a GIS framework. The inventory data used by HAZUS-MH can potentially become very large depending on the extent of the hazard coverage. In addition, the geographic extents and resolution could result in high volumes of input data. Comprehensive analysis of such data sometimes is not at all possible by employing a standard desktop system, and even if it is, it usually takes hours to days before the application can obtain any analytical results. Even for non-emergency response applications, where processing speed is not critical, spatial data processing routines run for extended periods of time.

For a wide range of large scale distributed computing applications from Geosciences,

the demand for resources varies significantly during the course of execution. While a set of dedicated resources for such applications could result in under-utilization most often, at other times the system could perform better by utilizing more resources than available. The emerging cloud platforms, such as Microsoft's Azure - with their potential for large scale computing and storage capabilities, easy accessibility by common users and scientists, on demand availability, easy maintenance, sustainability, and portability - have the promise to be the platform of choice for such GIS applications.

Some studies have been conducted to understand the extent of support that cloud computing can or cannot facilitate for large scale distributed scientific applications [4, 11, 12]. There are only a few projects from Geosciences that have been designed specially for cloud platform. Most relevant among these include ModisAzure project for download, re-projection, and reduction of satellite imagery [4, 11, 50], and Smart Sensors and Data Fusion applications project for ocean observation [51]. After an extensive review of literature on vector data based spatial overlay processing, we have found that

1. none of the existing projects employ cloud-computing for parallel or distributed spatial overlay analysis on vector data,

2. although parallel and distributed algorithms have been reported in literature for vector overlay computation (primarily in 1990s), there is very little background literature by ways of implementation projects and performance results even on traditional parallel and distributed machines, and

3. although both commercial and open-source projects are available for vector-data based spatial overlay processing, the state-of-the-art is desktop based computing, and none of them employ even a threaded version of any parallel algorithm for overlay analysis.

We have engineered *Crayons* system over Azure cloud with a parallel, open software architecture for traditional polygon overlay analysis. We believe *Crayons* to be the first cloud-based system for end-to-end spatial overlay processing on vector data.

In this chapter, we document the details of *Crayons* system, including meta-algorithm for carrying out the spatial overlay computation starting from two input GML files, their parsing, employing the bounding boxes of potentially overlapping polygons to determine the basic overlay tasks, partitioning the tasks among workers role processes, and melding the resulting polygons to produce the output GML file. Along with the documentation of software architecture of the *Crayons* system, we discuss how the distributed computing artifacts available in Azure cloud platform affected our design choices for *Crayons*. The atypical implementation issues encountered due to (i) the idiosyncrasies of Azure environment, and (ii) the third party clipper library for domain-specific code for solving basic spatial overlay, are discussed to provide roadmap for follow-up work. We study the timing characteristics of various phases of the meta-algorithm rigorously. Our thorough experiments and analysis, by means of three different design architectures for *Crayons*, give insights into the Azure cloud platform and points to the phases of the algorithm that are easily amenable to scalable speedups and some others that are not.

We have designed three different architectures of *Crayons*; these versions are labeled based on how the load is distributed among worker processors. In the first version, the web role acts as a master processor and sends tasks to the task pool. The worker roles, acting as traditional slave processors, fetch work from the task pool and perform the spatial processing. This version is referred as the version with *centralized load balancing* among workers. In the second and third version, we do not use a master processor that creates tasks for everyone. Instead, all of the worker processors create tasks and thus task creation process itself is distributed. These versions differ in the manner they share tasks among each other. In the first distributed version, referred as *distributed version with static load balancing*, the workers create and process tasks locally. There is no sharing of tasks of one processor with other processors in this version. In the final version, the workers create tasks and put these tasks in a shared task pool. After finishing the creation of tasks, the workers fetch the task from the task pool and process them similar to centralized load balanced version. This version is referred as the *distributed version with dynamic load balancing*.

Our three different architectures can be presented in an academic or commercial setting to educate students and developers on various design issues for cloud platform, especially when the underlying platform lacks the support for traditional distributed or parallel software infrastructures such as MPI and map-reduce. Moreover, we have developed *Crayons* using C# language in Microsoft Visual Studio environment making it further simplified for others to experiment with *Crayons* using familiar tools.

Our specific technical contributions are as follows:

- Engineering an end-to-end spatial overlay system by way of designing and implementing three partitioning and load balancing algorithms: (i) Centralized Dynamic Load Balancing, (ii) Distributed Static Load Balancing, and (iii) Distributed Dynamic Load Balancing (Section 4.3).

- Open architecture of *Crayons* for interoperability with any third party domain code (clipper library) for sequential execution of primitive overlay computation over two polygons (Section 4.2.4).

- Port of *Crayons* over a Linux cluster using MPI platform for (i) scenarios where the data is too sensitive to be stored on a cloud platform, and (ii) to facilitate porting of *Crayons* to systems with traditional parallel and distributed software architectures (Section 4.5.8).

- End-to-end speedup of more than 40x using input files with comparatively uniform load distribution, and more than 10x using input files with skewed load distribution, using 100 Azure processors for basic overlay computation (Section 4.5).

- Making *Crayons* available as an open-source project to be used as a reference architecture for introducing HPC and GIS application development on Azure cloud platform (Section 4.1).

The rest of this chapter is organized as follows: Section 4.1 describes our motivation behind the pioneering *Crayons* system. Section 4.2 reviews the literature and provides

background on GIS raster and vector data, various operations that define parallel overlay, and the tools used to implement these operations such as Windows Azure cloud platform and General Polygon Clipper (GPC) library. Section 4.3 describes our parallel Azure framework and its three flavors. Several key implementation related issues are discussed in Section 4.4. Our experimental results, load balancing and other experiments, and port to MPI are presented in Section 4.5. Section 4.6 concludes this chapter with comments on future work.

4.1 Motivation

An End-to-end GIS Solution: Currently, GIS scientists have no other alternative but to use desktop based sequential GIS system with typical runtimes in hours making it almost useless for real time policy decisions. This is not for the lack of individual parallel algorithms and associated techniques as is evident from the literature [47, 52–56]. It has been a long-standing question to effectively engineer all the pieces together due to the data intensive and irregular computational nature of GIS applications, as we discovered. With *Crayons* we have addressed these questions and engineered an effective system to fill this void.

On-demand Cloud Computing: Lee et al.[4] demonstrate the need for efficient parallel computation of GIS models with an example of operational hurricane track forecasting models used for prediction during hurricane Katrina in 2005. There were five different models applied for this forecasting five days in advance. With five and four days out, there was no agreement among those models. The models only agreed with ground truth after second and first day out. Computation of such time critical applications significantly affects the effectiveness of emergency response operations. Authors conclude that the only answer to this *scientific and operational grand challenge problem* is enormous computer power. However, it is not economically possible to dedicate the required amount of resources for this single purpose. Therefore 1. resources must be shared but available on-demand, 2. the platform should be scalable on-demand, and 3. resources should be easily accessible to

GIS scientists and policy decision makers (not necessarily computer-savvy) in a user friendly way over the web. These guidelines are mandatory for the proliferation of such GIS systems. Since a Grid or any large compute cluster cannot adapt to these guidelines, we chose to architect *Crayons* for cloud computing as cloud computing appears to be the only feasible platform.

Azure Cloud: HPC program development over the still-emerging Azure platform and continually changing APIs is very difficult and tedious even for experienced parallel programmers and algorithm developers. However, the importance of "blazing this trail" is recognized by experts as evidenced by the rare partnership between NSF and Microsoft in funding this Azure effort. Additionally, we chose Azure cloud platform over other rather mature cloud platforms as Azure platform provided us the opportunity to think-outside-the-box to devise an architecture for systems research for data and compute intensive scientific applications as it currently lacks support for traditional distributed computing design paradigms such as MPI or map-reduce. On the other hand, Azure's robust middleware APIs and artifact enable finer-grained task level fault tolerance (details in Section 4.3.1) which other clouds with system image level control cannot.

Collocated Data and Compute Services: It is of utmost importance for data intensive scientific applications, especially where data travels over a network, to be able to control/hide the latency for a smoother user interaction. Azure cloud platform supports *Affinity Groups* where the computing resources (hosted services) and storage resources (storage accounts) can be collocated in the same geographic location. The key idea behind an Azure Affinity Group is that if multiple services in a cloud subscription account need to work together - for instance, if a hosted service stores data in the Blob storage service, or Table storage service, or relies on the Queue storage service for workflow - then the hosted service and storage accounts can be organized within the same affinity group for optimal performance. Azure Affinity Group provides a best effort based control on data and compute collocation at regional level (US North Central, US South Central, Europe North, Europe

West, Asia East, and Asia Southeast). It is still far from the control that would be required for a fine-grained task scheduling in a distributed platform such as Azure cloud.

Fostering Educational Activities with *Crayons* System: *Crayons* is an open-source project with all three design architectures (see Section 4.3) hosted at an online repositorey, available under EPL (Eclipse Public License). The web application is hosted at Azure cloud with small sample GIS data sets for in-class demonstrations. Faculty and students can also install Azure simulator and download *Crayons* locally to execute and study *Crayons*' workflow. *Crayons* can also be presented as an example of an architecture design where the underlying platform does not support well-known parallel and distributed design paradigms such as MPI and map-reduce.

4.2 Background and Literature

4.2.1 Raster vs. Vector Data in GIS

Similar to pixels in an image, raster data represents the geographic space as an array of equally sized cells. Each cell has attributes associated with it to define the geographic features. Perceptibly, cells with same attribute values represent the same type of geographic features. Raster data can be stored as a matrix, where dimension of matrix depends on the number of bands that raster data is composed of.

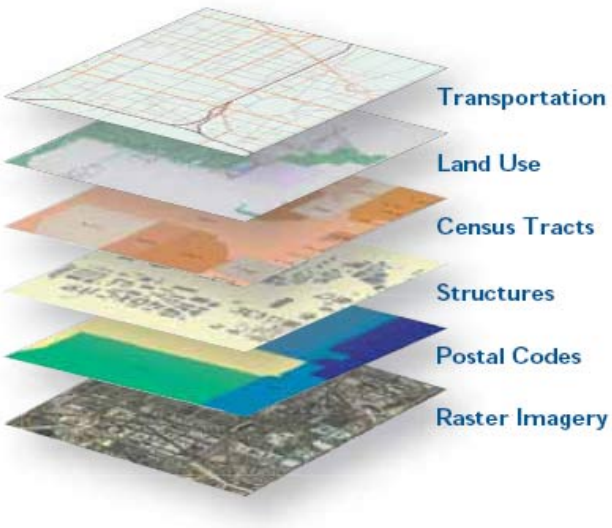


Figure 4.1 Real world data organized into thematic layers. Image courtesy: FPA [2]

Unlike raster data, vector data model represents geographic features as points, lines, and polygons. Geometric shapes in vector data model are defined by geographic

coordinates in a certain type of projection upon a geodetic datum. Similar to raster data, each geometric feature has attributes associated with it that describe the characteristics of the feature.

4.2.2 Spatial Overlay Operations

GIS collects and stores information in the form of thematic layers as shown in Figure 4.1. Since the data are referenced to geography, multiple layers from one location could overlay one another. GIS connects the location to each layer such as people to addresses, hurricane swath to rescue shelters, transportation services to road networks etc., to give a better understanding of how these independent layers interrelate with each other.

Both raster and vector data are capable of representing the same GIS information, although both have associated advantages and disadvantages. Moreover, the same feature could very well be represented in multiple ways. For instance, a county can be represented as a point feature in a small scale map, while in a larger scale map the same county can be described as a polygon.

In order to analyze the spatial relationships between sets of geographic features, within the same spatial scope in a map frame from the given layered spatial data set, all types of geographic features can be overlaid.

This spatial analysis differs significantly for raster and vector data. In raster data, overlay operation is straightforward as it involves arithmetic overlay computation, such as addition, subtraction, division, or multiplication of raster data layer matrices. However, spatial overlay operations over vector data are more complex. Spatial overlay operation over vector layers involves rebuilding topological relationships in the derived data, it is therefore normally known as topological overlay. For instance, when line segments from two thematic layers intersect, new vertices are formed. When polygons are created by overlaying two polygon layers, the new features may inherit the attributes of the spatially overlaid polygons. Since new topological framework is created from two or more existing topological networks of the input data layers, rebuilding of topological tables, such as the arc, node, polygon, can

be time consuming and CPU intensive.

4.2.3 Parallel Overlay Operations

Spatial vector data processing routines are widely used in geospatial analysis. There is only a little research reported in literature on high volume vector-vector or vector-raster overlay processing [47]. Since spatial overlay processing depends on the implementations of suboptimal algorithms [52–54], the processing costs can vary significantly based on number, size, and geometric complexity of the features being processed [57]. There has been extensive research in computational geometry that addressed scalability and parallel or out-of-core computation [55, 56]. Nevertheless, the application of this research in mainstream GIS has been limited [53, 54]. Some research exists for parallel implementations of vector analysis, showing gains in performance over sequential techniques [58–60] on classic parallel architectures and models, but none on the clouds.

4.2.4 Clipper Library

We have designed *Crayons* such that third party GIS libraries can be plugged in to facilitate reusability, interoperability, and accuracy based on user preferences. *Crayons* currently employs a third party clipper library called GPC [61] to delegate primitive geometrical operations. GPC library is an implementation of polygon overlay methods described in Vatti et. al. [62]; it can support subject and clip polygons that are convex or concave, self-intersecting, contain holes, or are comprised of several disjoint contours. Moreover, the original algorithm in [62] has been extended such that GPC library allows source polygons to have horizontal edges. GPC library only supports four types of overlay operations namely intersection, exclusive-or, union, and difference. The output may take the form of polygon outlines or tristrips.

4.3 Our Parallel Azure Framework Design

We have spent considerable effort analyzing the still-emerging Azure platform's nitty-gritty to gain insights into Azure framework, parallel reading and writing from and to cloud storage, and load balancing. In the process, we created three different architectures for *Crayons* to thoroughly test the Azure platform's design artifacts and to carve path for future development.

Crayons contains one instance of Azure web role that serves as the user interface and delegates work to worker role instances. For background processing of spatial overlay operations, multiple instances of Azure worker roles are employed. The number of worker role instances was varied from 1 through 100 to test the scalability. To store GIS files in GML format persistently *Crayons* uses Blob storage, and to store messages needed for communication between web role and worker roles Queue storage is employed.

Crayons' framework has a three to four-step workflow (meta-algorithm) with multi-stage pipelining between producers and consumers. The multi-stage pipelining can be seen in action during task creation and task processing. The task processing starts as soon as there is a task in the task pool and a worker waiting for task. This gives us the benefit of not waiting for the previous phase to complete entirely before the next phase can start processing, similar to a traditional manufacturing assembly line.

4.3.1 *Crayons*' Architecture with Centralized Dynamic Load Balancing

Figure 4.2 shows the architectural diagram of *Crayons* with centralized load balanced version employing an extra large virtual machine (VM) (i.e., 8 core machine, see Table **??**) as the centralized task producer. End users have the option to upload their input files in GML format to cloud or to operate on the existing files. Since uploading is a trivial process, for the sake of simplicity to understand the workflow we will assume that the files are already available in the cloud storage. The entire workflow for this architecture is divided into three steps as defined below:

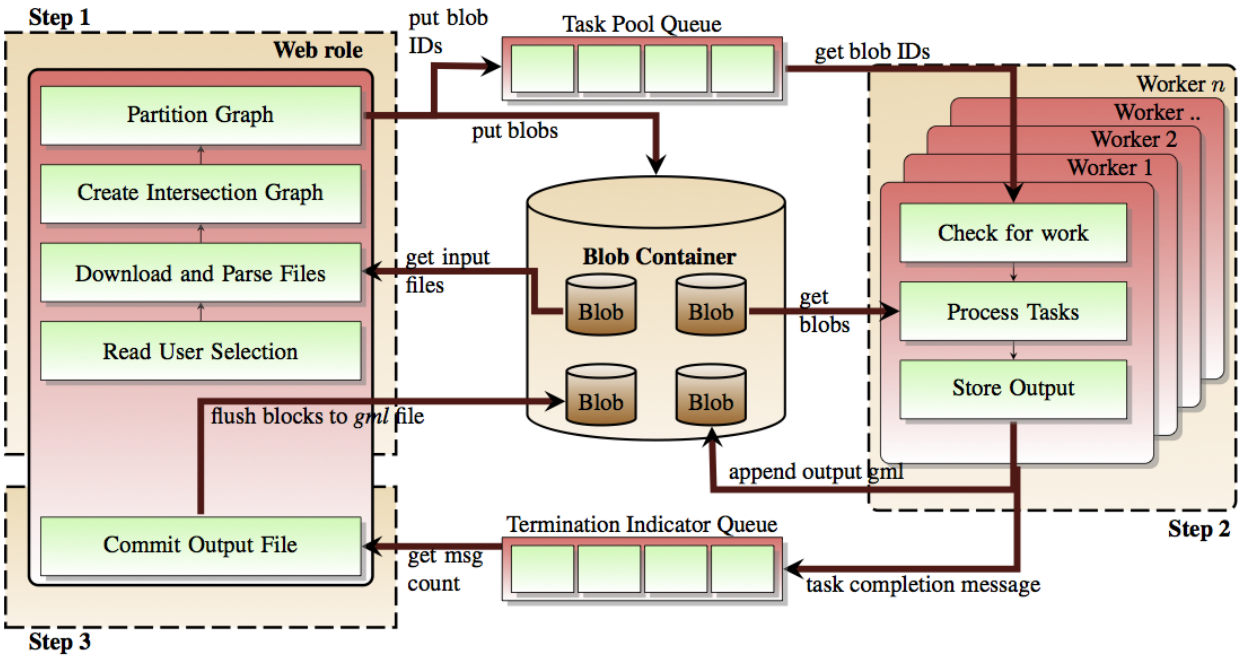


Figure 4.2 *Crayons*' Centralized architecture

I. The web role presents the interface with a list of data sets (GML files) available to be processed along with the supported operations. The user selects the GML files to be processed along with the spatial operation to be performed on these files. First of these two selected files is treated as the base layer and the second file is treated as the overlay layer. The web role immediately starts downloading the files from the Azure cloud storage and translates (parses) the features (polygons) from the input GML files into C# objects.

Since spatial overlay operations are computationally expensive, it is wise to prune the set of polygon pairs needed to be processed together. In order to create this intersection graph, *Crayons* finds each overlay polygon that can potentially intersect with the given base polygon and only performs spatial operation on these pairs. As shown in Algorithm 6 this is achieved using the coordinates of bounding boxes generated during parsing of input files. Intersection graph creation currently is based on sorting the polygons with $\Omega(n \log n)$ cost [60]. This phase can potentially be improved by employing R-Trees [63] in a later version.

Intersection graph defines one-to-many relationship between the set of base polygons

[1]A bounding box is represented using bottom-left and top-right points with X and Y coordinates.

Algorithm 6 Algorithm to create polygon intersection graph (similar approach to [60])

INPUT: Set of Base Layer polygon S_b and Set of Overlay Layer polygon S_o
OUTPUT: Intersection Graph (V,E), where V is set of polygons and E is edges among polygons with intersecting bounding boxes.

```
Parallel Merge Sort set S_o of overlay polygons based on X co-ordinates of bounding boxes[1]
for all base polygon B_i in set S_b of base polygons do
  find S_x ⊂ S_o such that B_i intersects with all elements of S_x over X co-ordinate
  for all overlay polygon O_j in S_x do
    if B_i intersects O_j over Y co-ordinate then
      Create Link between O_j and B_i
    end if
  end for
end for
```

and overlay polygons. To create an independent task, one polygon from base layer and all intersecting polygons from overlay layer are merged together as a task and stored in the cloud storage as a Blob. The web role converts the C#'s polygon objects belonging to a task to their GML representation before the task gets stored in the Blob storage. We prefer in-house serialization against C#'s serialization library to avoid excessive metadata required to convert an object to string (details in Section 4.4.1).

Each task is given a unique ID, this id is communicated to the worker roles using a message over a Queue that serves as a shared task pool (see Figure 4.2) among workers and thus facilitates dynamic load balancing. Queue storage mechanism provided by Azure platform comes handy here to implement task based parallelism and for fault tolerance as discussed later in this section.

II. Worker roles continuously check the shared task pool (Queue) for new tasks. Since this can throttle the Queue storage - with a limit to support a maximum of 500 requests per second - if there is no message in the Queue we let a worker sleep for a few seconds before sending next request. However, if there is a task (message) in the shared task pool, the worker reads the message and consequently hides it from other workers, downloads the Blob with ID stored in this message, converts the content of the downloaded Blob to get the original base and overlay polygon objects back (deserialization), and performs the

spatial overlay operation by passing a pair of base polygon and one overlay polygon at a time to *GPC* library for sequential processing.

GPC library returns the resultant feature as a C# polygon object that is converted to its equivalent GML representation and appended as a block to the resultant Blob stored in the cloud storage. Azure API *PutBlock* is used to achieve parallel writing to the output Blob. This API facilitates the creation of a Blob by appending blocks to it in parallel and if the sequence of the features is not critical, which is the case here, this API can significantly improve the performance. After each task is processed the corresponding worker role permanently deletes the message related to this task from the task pool Queue. Additionally, each worker role puts a message on the termination indicator queue to indicate successful processing of the task.

III. The web role keeps checking the number of messages in the termination indicator queue to update the user interface with the current progress of the operation. Logically, when all of the tasks have been processed the number of messages in the termination indicator queue will match the number of base polygons. When this happens, the web role commits the resultant Blob and flushes it as a persistent Blob in the Blob storage. The resultant Blob becomes available for downloading or further processing, user interface is also updated with the URI of resultant Blob. To commit a Blob created using blocks the Azure API *PutBlockList* is used. In order to use *PutBlockList* it is necessary to provide the list of blocks to be committed, this list is maintained at the cloud end and can be downloaded by the web role by using another Azure API *GetBlockList*. The output Blob's *URI* (uniform resource indicator) is presented to the user for downloading or further processing.

The Queue storage mechanism provided by Azure platform comes handy for fault tolerance during processing. After a worker role reads a message from the task pool, the message disappears from the task pool for other worker roles and is subsequently deleted by the worker role after the processing ends successfully. In the event of a failure, the message does not get deleted and appears in the Queue after a stipulated amount of time.

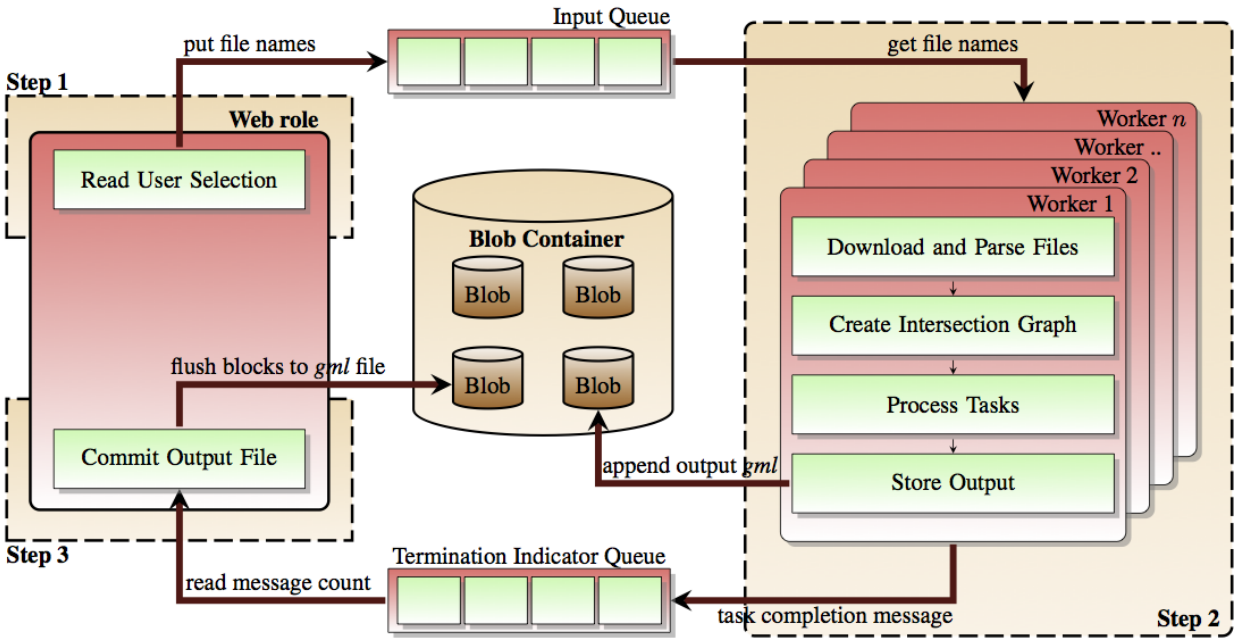


Figure 4.3 *Crayons*' architecture with distributed static load balancing

One significant disadvantage of the centralized version of *Crayons* is that the subprocesses of file handling, task creation, and storing tasks to Blob storage are handled by a single virtual machine (VM). The worker roles keep waiting idly for work until the task creation begins and the tasks IDs are available in the task pool. Moreover, regardless of the size of the VM, with the increasing number of worker roles there will be a demand-supply imbalance that will negatively affect the performance. Therefore, we created the distributed versions of *Crayons* for superior load balancing and scalability.

4.3.2 *Crayons*' Architecture with Distributed Static Load Balancing

Figure 4.3 shows one of the two distributed versions of *Crayons*' parallel architecture to parallelize the subprocesses of intersection graph construction and task creation. Here, the web role is a small sized virtual machine, i.e., a single core machine, as all computationally intensive tasks are handled by worker roles. The entire workflow for this version is a three-step process as described below:

I. Similar to the centralized version, the web role presents the interface with a list of data sets (GML files) available to be processed along with the supported operations. The user selects the GML files to be processed and the spatial operation to be performed on these files. Instead of processing the files by itself, web role writes the names of the files along with the operation to be processed to the input queue.

II. The worker roles get the message out of the input queue and download the corresponding input GML files from the cloud storage. Unlike the case of task pool queue in centralized version, workers use an Azure API *PeekMessage* so that the message does not become invisible to other workers.

Although it appears to be an overkill for all workers to download the same files, it does not affect the performance much as in the earlier case (Section 4.3.1) the worker roles kept waiting idly on the web role to finish parsing, intersection graph creation, and pushing tasks to input queue. However, parallel download of the same file by multiple workers does cause a little contention at cloud storage and the parsing phase ends at different time stamps for different workers. Details of this are discussed in Section 4.5.

In order to distribute work among worker role instances each worker is assigned work based on its instance ID. Once the GML files are downloaded and parsed, the workers create independent tasks only for their portion of base layer polygons. This obviates any need for communication among worker roles.

The tasks are created and stored in the Blob storage in a fashion similar to the case of centralized version (Section 4.3.1). In contrast to the previous version, each worker role keeps track of only its own task IDs and stores them in local memory rather than storing it in a shared task pool. The advantage of this technique is that it saves the time spent in reading and writing messages from and to the Queue storage. On the other hand, if the input data files have skewed task distribution some workers will have considerably more work compared to others. The workers with lighter work loads will finish earlier and will wait idly resulting in wasted CPU cycles.

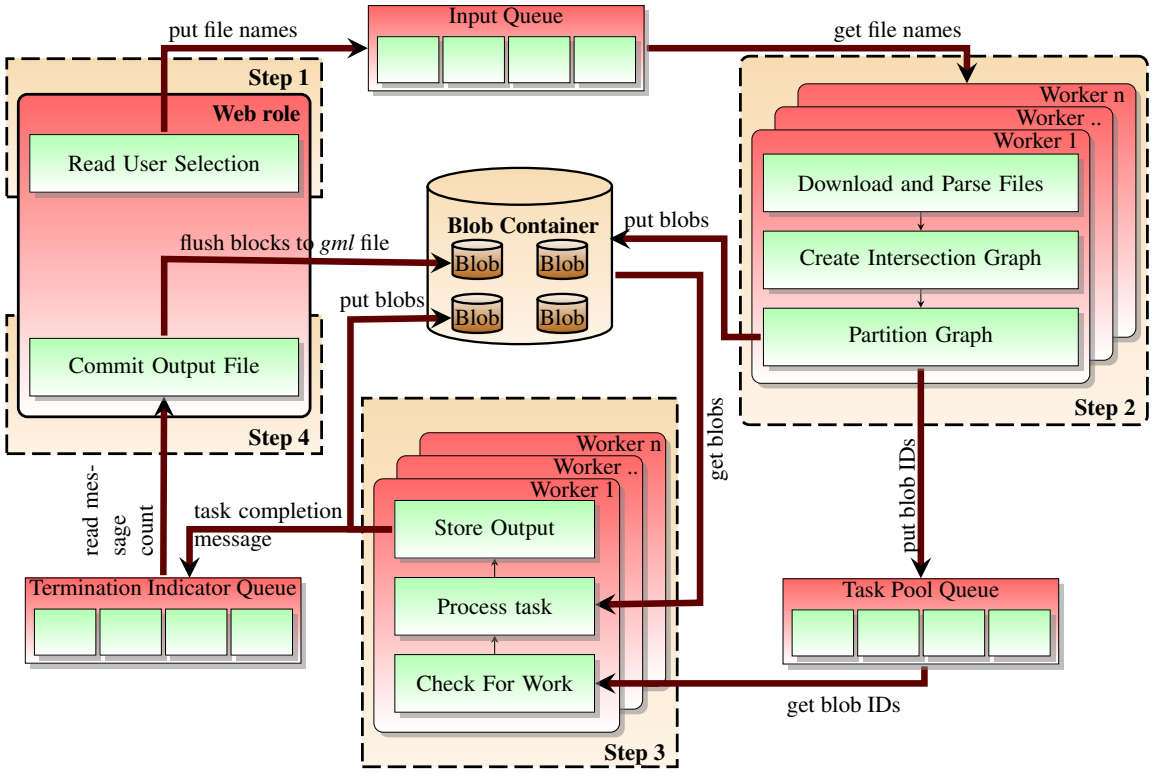


Figure 4.4 *Crayons* architecture with distributed dynamic load balancing

The task processing and storing the outputs in the Blob storage is similar to the centralized version. Moreover, a message is stored on the termination indicator queue to signal the successful processing of one task.

III. This step is also similar to that of centralized version. The web role updates the user interface with progress based on number of messages in termination indicator queue and finally flushes the output Blob to the Blob storage. The output Blob's *URI* (uniform resource indicator) is presented to the user for downloading or further processing.

4.3.3 *Crayons*' Architecture with Distributed Dynamic Load Balancing

Although *Crayons*' distributed architecture with static load balancing is superior to the centralized version, it suffers from two significant problems. First and foremost, if a worker role fails during execution, it needs to download and parse the files again as the task IDs were held locally and thus were lost too. Secondly, in case of skewed task distribution in

input GML files it is possible that in the worst case all of the spatial overlay processing happens at a single worker role instance while other worker role instances wait idly. This is possible if all of the intersecting polygons from overlay layers were only intersecting with a small subset of base layer polygons. To overcome both of these shortcomings, we created a new version of *Crayons* (Figure 4.4) that employs (i) parallel execution of subprocesses of file handling, task creation, and storing of tasks in Blob storage, similar to distributed version (Section 4.3.2), and (ii) dynamic load balancing similar to centralized version (Section 4.3.1). The entire workflow of this version is divided into four steps.

I. The first step is similar to that of *Crayons*' architecture with distributed static load balancing. The user selects the files to be processed and the operation to be performed. The web role puts this information on the input queue.

II. In this step, the worker roles download the input files, parse them, create the intersection graph to find the independent tasks. Then each worker role shares the tasks it created among all the worker roles. Unlike the version with static load balancing, the worker role instances here do not store the task IDs of the tasks that they create locally. Instead, they store the task IDs in a task pool (Figure 4.4) similar to centralized version.

III. As the workers finish task creation, they keep fetching work from the task pool and go on processing all the tasks. The advantage of this approach over the approach with storing local IDs is that the worker role instances can also process the work of other worker role instances and hence achieve improved performance even with skewed input data.

IV. This step is exactly similar to previous two versions. The web role checks the number of messages in the termination indicator queue and when this number matches the total number of tasks, the web role flushes out the blocks to the Blob storage and the output file becomes available for further processing and download.

4.4 Engineering Details

Our primary contribution is that *Crayons* system is an "engineering" feat to create an "Azure" cloud based overlay processing system with an open architecture. During the course of engineering *Crayons* system we ran into multiple issues caused variously by the development tools and platforms used. In this section we address those issues and suggest our solutions to provide a roadmap for readers interested in development on Azure platform. It is our strong belief that this discussion will help readers make informed design choices.

4.4.1 Azure-specific Issues

HPC program development over the still-emerging Azure platform is very difficult and tedious even for experienced parallel programmers and algorithm developers. Following are the issues that affected our design choices and are essential to know beforehand for the development of large scale scientific applications on azure.

Table vs. Blob Azure Table storage provides a service to store large amount of data that needs additional structure. Table storage can be queried to retrieve data based on the underlying structure. This looks promising and thus was the first choice for *Crayons* system. The idea was to store the GIS features (polygons and holes) in an Azure Table storage to enable easy spatial querying and static task assignment to workers virtually arranged in a 2D grid. However, a table can only store entities that are a maximum of 1 MB in size. Since *Crayons* handles and stores tasks that are usually larger than 1 MB, table storage was realized not to be a good fit in this case. Although large tasks can be broken into multiple parts, each less than 1 MB, that would make the code complex without any necessary performance gain. Therefore, we chose to proceed with Blob storage that can handle Blobs of size up to 64 MB or block Blobs of size up to 200 GB.

Queues - FIFO Behavior *Crayons* uses Queue storage service of Azure platform to communicate between web role and worker roles. Azure Queue storage differs from the

traditional Queue data structure as it lacks the ability to guarantee a FIFO operation. This lack of guarantee for FIFO operation can create issues if a Queue is to be used to signal a special event. For instance, if web role wants to append a message at the end of the task queue to signal the end of work, it might not work as expected. Since FIFO is not guaranteed, the worker roles might read this message before the actual messages for tasks and hence quit processing while there is work in the task pool. This is the reason why we had to create a dedicated termination indicator queue where worker roles send messages to indicate successful completion of tasks. The web role keeps track of the number of messages in this Queue and thus knows the current progress of the processing.

Serialization vs. GML Representation Azure Blob storage can be used to store a polygon object from C# only after it has been represented as bytes or a long string of text. Microsoft C#.Net provides a serialization library that can take an object and convert it into a byte array or a string that can later be de-serialized and restored back as the original object. Although, serialization of objects significantly simplifies the distribution of objects among worker and web roles as objects can now be stored as Blobs, the serialization library creates a graph tree of the objects and adds a significant amount of metadata to ensure that the object can be created back from the serialized string and thus is a rather time consuming process. Since *Crayons* deals with an enormously large number of polygons this process costed us a lot of chargeable compute hours as well as storage hours on Azure platform. Nevertheless, it is inevitable to serialize the objects as the polygon objects must be distributed among worker roles through Blob storage.

To alleviate this problem we created an in-house serialization library that incorporates the information needed to be stored and thus can convert an object to its GML representation. To convert an entire task to a string all the polygons in the task are converted to their respective GML representations and then concatenated in such a way that the base layer is the first polygon. With this information the polygons can be converted to strings and vice versa.

Local Azure Simulator vs. Azure Cloud Environment Microsoft Azure provides a simulator that can be installed on a personal computer so that Azure based applications can be debugged locally. The development cycle for Azure cloud based applications starts with a local desktop-based development of that application on an Azure simulator and ends with the application deployed to the cloud. Interestingly, we found that the cloud platform does not necessarily replicate the local Windows OS and .Net platform appropriately. In our case, the issue was absence of a standard Windows dynamic link library (dll) file named *msvcrt100.dll* on Azure cloud platform; the application started executing as expected once this dll file was manually packed inside the package that was deployed to the Azure cloud. In order to resolve such errors the best method is to login to the remote VM through remote desktop connection and checking the local error logs at that VM.

4.4.2 Large Data Sets and Concurrency Control Mechanism

In order to make it easier for developers, Microsoft .Net platform supports multiple parallel directives such as *Parallel.For* loop and others. These mechanisms try to spawn as many threads as possible based on the configuration of underlying system. The Azure SDK does not keep track of how many threads can actually query the cloud system and thus can throttle the cloud storage by sending too many requests. In our case, the application worked fine with the small data sets but when the system was loaded with large data sets there were *unhandled exceptions* being thrown that would not let the program continue. It took us weeks of debugging to find the cause of the problem and subsequently the solution. The solution was not to let the system decide the number of threads but to control the concurrency by creating a thread pool with limited number of threads. Each thread would spawn, perform the expected operation, and go back to the pool.

4.4.3 Clipper Library

General Polygon Clipper (GPC) library is a well-known library for performing spatial overlay operations. However, applicability of GPC library as a full-fledged GIS overlay library

is limited due to a few design choices. First and foremost, the GPC library only supports four operations - intersection, exclusive-or (XOR), union, and difference, while GIS overlay operation should be able to test the source polygons for an extensive set of relationship such as *Equals*, *Crosses*, *Within*, *Contains*, *Disjoint*, *Touches*, and *Overlap* in addition to the operations supported by GPC library.

Secondly, although GPC library is capable of handling polygons that contain holes, the resulting polygon does not discriminate between a polygon contour and a hole. It is left to the user to compute that information by some means. Murta et. al. [61] suggest that to associate holes H_1, H_2 ... H_n with external contours E_1, E_2 ... E_m use the clipper to compute the difference of the ith hole H_i and each external contour E_j, for all j from 1 to m. Any difference which gives an empty result indicates that hole i lies within external contour j. This clearly adds $O(mn)$ complexity where m and n are the number of resultant polygons and holes.

If the GIS overlay operation is part of a multi-step process, output of current phase might be input to other phase and thus causing erroneous results. Moreover, it does not maintain all of the attributes of polygons, such as polygon ID and dimension information among others. We had to store this information locally to copy it over to the resultant polygon after the library returns the output. When the output of GPC library was compared with that from state-of-the-art GIS solution ArcGIS, it was found that the output polygon boundaries had a little deviation. A small number of vertices (1-2%) were missing from the output too. However, the overall timing characteristics of the *Crayons* system should hold for intersection and related operations, and our open system design would allow us to substitute the GPC library with others.

4.5 Performance of Crayons System

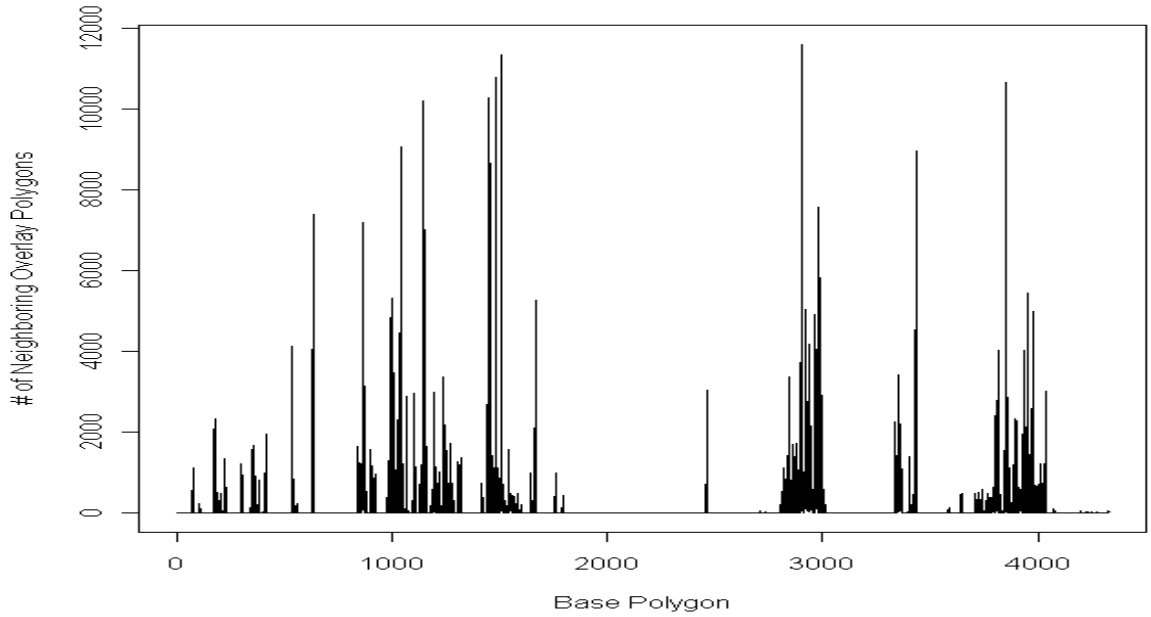
4.5.1 Load Balancing and *Crayons* Pipeline

Windows Azure can be configured with various VM sizes and number of instances of these VMs. These configurations significantly influence the amount of load balance that can be afforded by an application.

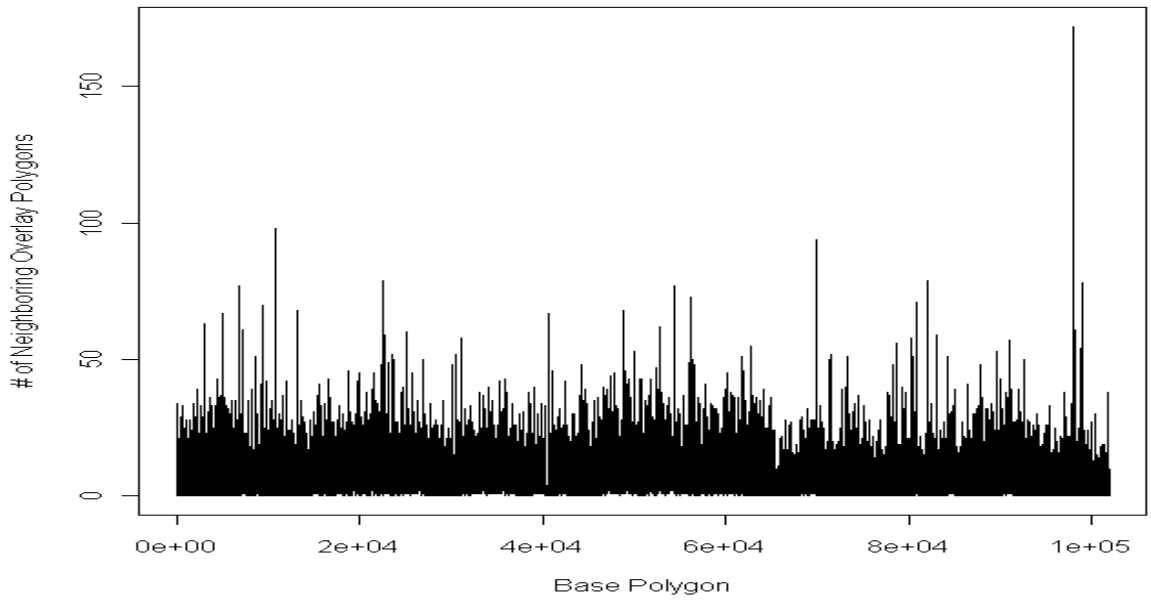
1. VM Size: Virtual machine sizes affect the configuration of the underlying processor on which a VM is going to run. The Azure cloud systems are typically AMD processors running at 1.5-1.7 GHz of clock frequency. Table **??** lists the configurations of virtual machines available to be used.

2. Number of Instances: Number of instances dictate how many distributed virtual machines will be allocated to a system. Applications accrue charges based on number of CPU hours used and hence the perfect blend of performance requirement and load balancing is inevitable to realize cost-effective performance. For the centralized version there is only one producer while the number of consumers was kept increasing to test scalability. Thus we decided to use an extra large VM for the producer and small VM size for the worker role instances. In both distributed load balanced versions, since there are no computationally intensive operations executing on the producer end it is a VM of small size. The worker role instances also do not use parallelism as we delegate the task of overlay to *GPC* library, which is sequential, hence we used small size VM for worker roles too.

We have a maximum quota of 100 cores that we can employ for our experiments. For the centralized distributed version of *Crayons*, 1 core is used by the user interface process, 8 cores are used by the instance that acts as the producer and rest 91 cores are used by the consumers. For the sake of fair comparison, we continue to utilize a maximum of 91 cores for worker role instances (consumers) in both distributed load balanced versions too.



(a) Skewed load distribution for smaller data set



(b) Comparatively uniform load distribution for larger data set

Figure 4.5 Load distribution plots for the data sets used for experiments

4.5.2 Input GML Files

We have used two different sets of input GML files for benchmarking *Crayons* system. Unless otherwise stated, all benchmarking has been performed over file size of 770 MB for the first layer containing 465,940 polygons and 16 MB for the other layer containing 4332 polygons - the output data file contains 502,674 polygons (***small data set***). The second set of GML files contains 101,860 polygons (242 MB) in the first layer and 128,682 polygons (318 MB) in the other layer (***large data set***). Total number of independent tasks was 4332 and 101,860 for the first and second data sets, corresponding to the respective number of base polygons. This is why we also call the first set smaller although it contains more polygons and takes more space than the second set.

Figure 4.5 shows the distribution of load between two data sets. The x axis is simply the base polygon and the y axis is the number of polygons from the overlay layer that can potentially intersect with the corresponding base polygon. Figure 4.5(a) shows how load is skewed in the smaller data set. Some polygons have no intersecting polygons while some have more than 10k intersecting polygons. The large load, as shown in Figure 4.5(b) is comparatively better distributed with maximum number of intersecting polygons mostly below 100.

The second set of files result in a better overall speedup due to better load distribution, but we have chosen to use first set of files for rigorous benchmarking of *Crayons* as it gives better insights into both *Crayons*' and Azure platform's scalability capabilities especially when load balance is skewed. Nevertheless, we report the performance of the dynamic load balanced version and speedup of *Crayons* using the second set of files too.

4.5.3 End-to-end Speedups over Small, Skewed Data Set

Figure 4.6 shows the speedup of *Crayons* system. The baseline timing is calculated over the distributed static version with only one worker role. Recalling from Section 4.3.2, this version does not store messages in the Queue and thus avoids that overhead. Moreover, since we are using a single worker with one core (small sized VM) the processing is sequential. We

preferred this version for sequential timing, rather than running Crayons locally, as this also ensures that the underlying machines for all of the experiments have similar configurations.

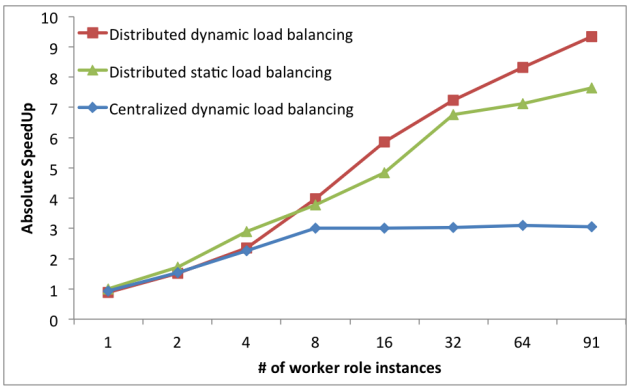


Figure 4.6 Speedup of *Crayons* system for small, skewed data set

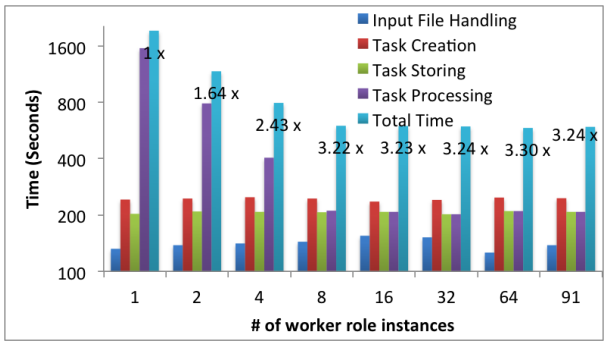
The end-to-end 1-processor time (the process of taking two GML files as input, performing overlay processing, and saving the output as a GML file) was 1803 seconds using *Crayons'* distributed version with static load balancing. The overall end-to-end acceleration of *Crayons* system starting from small set of input GML files to producing output GML files is about more than 9x as shown in Figure 4.6.

It can be clearly seen from Figure 4.6 that both of the distributed load balanced versions scale better than the centralized load balanced version. The reason, as discussed in Section 4.3.1 previously, is the demand-supply imbalance due to only one VM working as a producer while the number of consumers keep increasing for the centralized load balanced version.

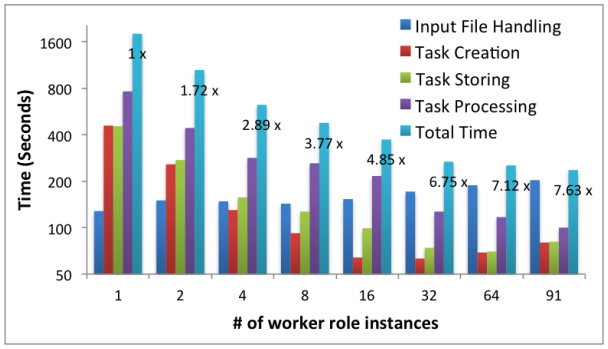
The reason for saturation of distributed load balanced versions is the inherent bottlenecks prevalent in Azure platform including simultaneous file download, contention of task queues, and parallel access to Blob storage. Due to these inherent bottlenecks, scaling of such systems on Azure platform will be challenging.

4.5.4 Timing Characteristics over Small Data Set

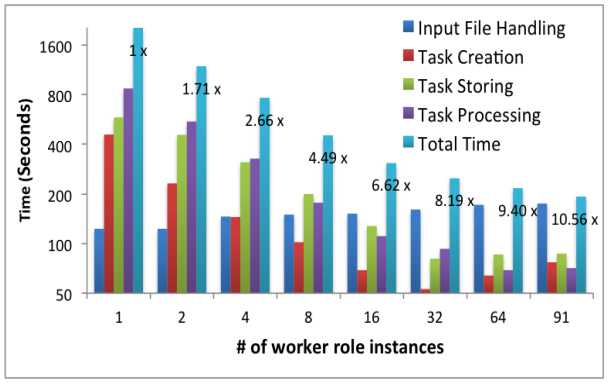
Figure 4.7 is a representation of maximum time taken by individual *Crayons* modules for all three architectures. The reported subprocess timings represent the time taken from first starting instance of that subprocess at any worker to the last finishing instance of that subprocess at any worker. For instance, if the subprocess parsing started first at time stamp t_a at worker role instance W_x then t_a is the start time for our experiment. Similarly,



(a) Centralized Dynamic Load Balancing



(b) Distributed Static Load Balancing



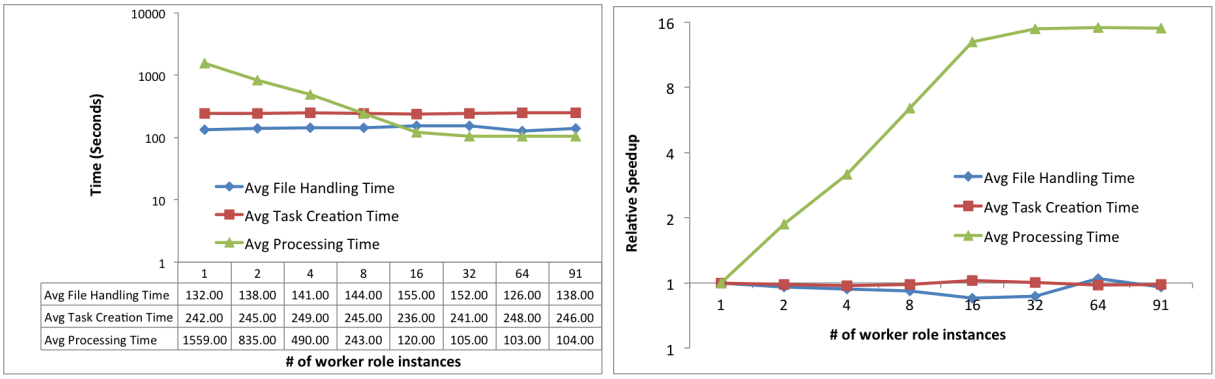
(c) Distributed Dynamic Load Balancing

Figure 4.7 Execution times for subprocess and end-to-end speedup over small data set

if worker role instance W_y was the last one to finish parsing at time stamp t_b, then the difference between t_b and t_a is the reported time. It is quite possible that workers other than W_y finished this subprocess earlier.

Lest the reader might confuse the total time as the sum of all subprocesses, it is important to note that the total time is the end-to-end time taken by the entire overlay operation and is less than the sum because *Crayons* leverages the pipelining of subprocesses.

***Crayons* with Centralized Dynamic Load Balancing** Figure 4.7(a) shows the time taken by subprocesses for the centralized dynamic load balanced version. Since Input File handling (downloading and parsing), Task Creation (intersection graph construction and task packaging), and Task Storing (insertion of tasks into Blob storage) are all done by one VM, there are minor fluctuations in these subprocesses. The reason for such minor



	1	2	4	8	16	32	64	91
Avg File Handling Time	132.00	138.00	141.00	144.00	155.00	152.00	126.00	138.00
Avg Task Creation Time	242.00	245.00	249.00	245.00	236.00	241.00	248.00	246.00
Avg Processing Time	1559.00	835.00	490.00	243.00	120.00	105.00	103.00	104.00

(a) Average execution time

(b) Relative Speedup

Figure 4.8 Centralized dynamic load balancing

fluctuations is the difference in CPUs (VMs) allocated during different runs and network latency to transfer data to all worker roles.

Number of worker role instances directly affect the timing characteristic of task processing subprocess. It can be seen from Figure 4.8 that the time taken by *Crayons* reduces with increasing number of worker role instances only up to 16 instances, and then it does not scale any further. Since web role is the only producer, increasing the number of worker roles (consumers of tasks) beyond 16 results in starvation.

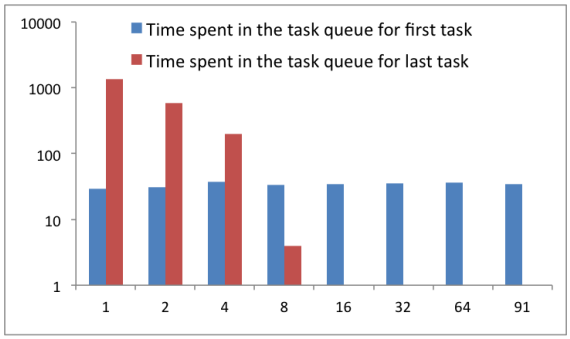
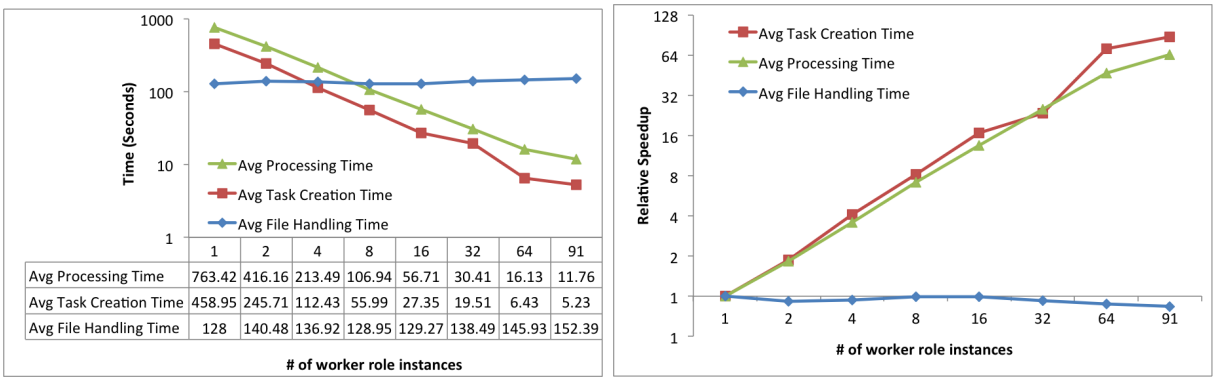


Figure 4.9 Task idling time (queuing time in the task queue) for *Crayons* version with centralized dynamic load balancing

This is further illustrated in Figure 4.9. It can be seen that up to 8 worker role instances the last message to be processed stayed in the Queue for some time. This means the workers were never starving as there was always more work than can be consumed, in the Queue. However, for the case of more than 8 workers, the tasks got consumed as soon as they were produced and hence never had a chance to wait in the Queue, i.e., there was either exactly as much work as can be consumed or less, but not more. Mathematically, the relationship



	1	2	4	8	16	32	64	91
Avg Processing Time	763.42	416.16	213.49	106.94	56.71	30.41	16.13	11.76
Avg Task Creation Time	458.95	245.71	112.43	55.99	27.35	19.51	6.43	5.23
Avg File Handling Time	128	140.48	136.92	128.95	129.27	138.49	145.93	152.39

(a) Average execution time

(b) Relative speedup

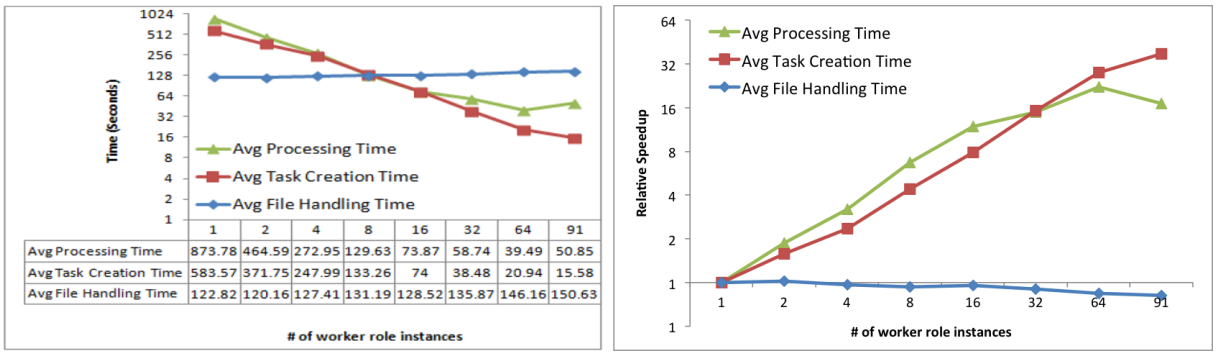
Figure 4.10 Distributed static load balancing

between number of tasks produced (P_t) by the producer and the number of tasks that can be consumed (C_t) by worker roles (W) can be described as

$$C_t \begin{cases} \leq P_t, \text{if W} \leq 16 \\ > P_t, \text{if W} > 16 \end{cases}$$

The timings shown in Figure 4.8 shows the individual timings for subprocesses for each worker role. The timings support the claim above as after 16 workers there is barely any gain in processing speedup as the number of worker role instances increases.

The reported times are the average times recorded by calculating the time for each of the three subprocesses (file handling, intersection graph creation, and processing the tasks using *GPC* library) for each worker role instance and then taking an average. As can be seen from Figure 4.8(b), the subprocess of task processing scales for 32 worker role instances giving a relative speedup of 14x. Since there is only one VM working on file handling and graph creation these timings stay almost constant throughout the experiments. The small variation is due to network latency and is likely a factor of traffic other than *Crayons'* as the data transfer over the network.



	1	2	4	8	16	32	64	91
Avg Processing Time	873.78	464.59	272.95	129.63	73.87	58.74	39.49	50.85
Avg Task Creation Time	583.57	371.75	247.99	133.26	74	38.48	20.94	15.58
Avg File Handling Time	122.82	120.16	127.41	131.19	128.52	135.87	146.16	150.63

(a) Average execution time

(b) Relative speedup

Figure 4.11 Distributed dynamic load balancing

***Crayons* with distributed static load balancing** Figure 4.7(b) shows the time taken by subprocesses for distributed static load balancing version. The distributed static load balanced version does not utilize the task pool as the workers store the IDs of tasks locally. The performance of this version can vary significantly with the work distribution among the input data files. This version will perform better with load balanced input data because it saves a considerable amount of time by avoiding the overhead of writing to the Queue storage (task pool). However, if the load is unbalanced this version will be only as good as the worker with the largest amount of work.

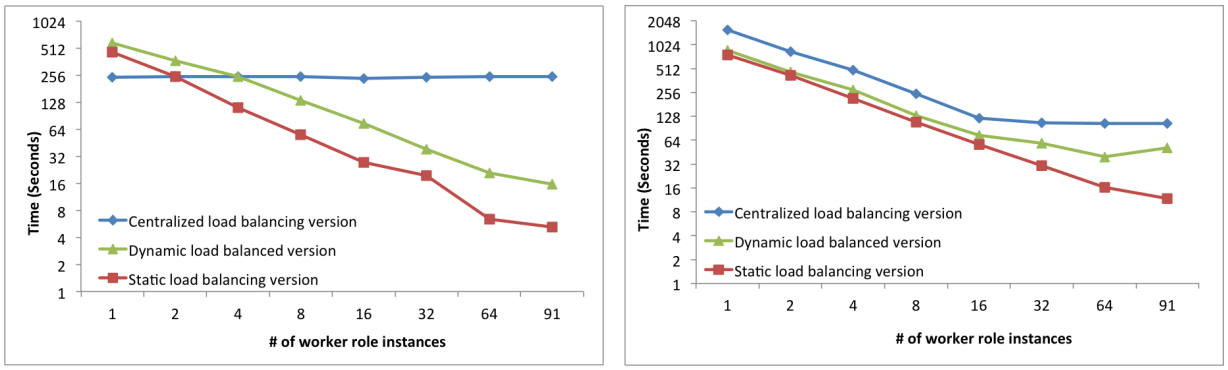
Figure 4.10 reflects the timing breakdown of subprocesses for worker roles for distributed static load balancing version. These timings are also the average timings as discussed previously in Section 4.5.4. Interestingly, the average execution time for task processing subprocess (average processing time) scales till 91 workers demonstrating the excellent scalability of *Crayons* system. Average file handling time is dependent on network latency, it varies from one execution to other even when every other parameter is the same.

The average time for task creation in this version is more than the centralized load balanced version because here we employ a small size VM, and thus only one thread is available, compared to the extra large sized VM employed by centralized version. This makes the parallel merge sort process (see Algorithm 6) slower for this version.

***Crayons* with Distributed Dynamic Load Balancing** Figure 4.7(c) demonstrates the timing characteristics of subprocesses for distributed dynamic load balanced version. This version is superior to centralized load balanced version as the work is shared among all the workers. Instead of only web role behaving like a producer, each worker role produces tasks for other worker roles. Impressively, this version scales till 91 worker role instances compared to 16 for centralized load balanced version and 32 for distributed static load balanced version. As can be seen by the Figure 4.7(c) the parsing time for this version keeps increasing with increasing number of worker role instances and hence parallel reading of input files from cloud storage becomes a bottleneck that subsequently affects other subprocesses negatively.

Figure 4.11 illustrates the average timings and speedup for various phases of distributed dynamic load balanced version. The average time for task creation, similar to distributed version with static load balancing, keeps decreasing with increasing number of worker role instances. The task creation and processing time for this version is greater than the distributed version with static load balancing as this version requires additional reading from and writing to Queue storage. The average processing time saturates at 64 worker role instances and hence increasing the number of instances beyond that only adds to the overhead of writing and reading from the Queue storage. This is the reason why the relative speedup for the subprocess of task processing drops a little for the case of 91 worker role instances.

Comparison of Various *Crayons*' Architectures Figure 4.12 compares the above three architectures based on the average time taken by the subprocesses of task creation and processing. We have chosen not to include the file handling process as the results are always almost constant (discounting the effect of network latency) for all three versions. It can be seen that the version with distributed static load balancing is superior to rest of the two versions. Interestingly, version with distributed static load balancing does not outperform the version with distributed dynamic load balancing when compared against end-to-end speedup shown in Figure 4.6. There are two points that this experiments has clarified.



(a) Average task creation time

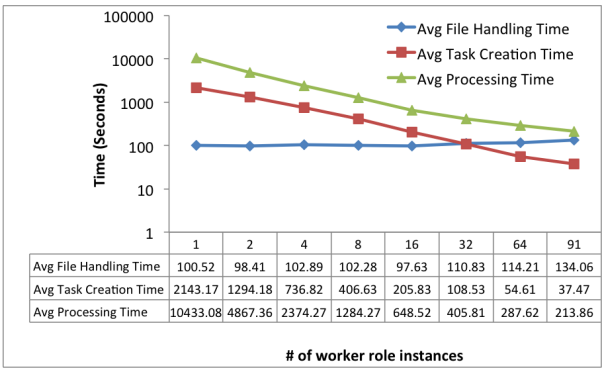
(b) Average processing time

Figure 4.12 Comparison across different versions of *Crayons* system

Firstly, the load imbalance among workers plays a big role in deciding which version of *Crayons* architecture is suitable for an application. This is the reason why the version with distributed static load balancing is outperformed by the version with distributed dynamic load balancing in the end-to-end time comparison. Secondly, the overhead of reading from and writing to Queue storage, further discussed in Section 4.5.6, has significantly plagued the performance of the distributed version with dynamic load balancing.

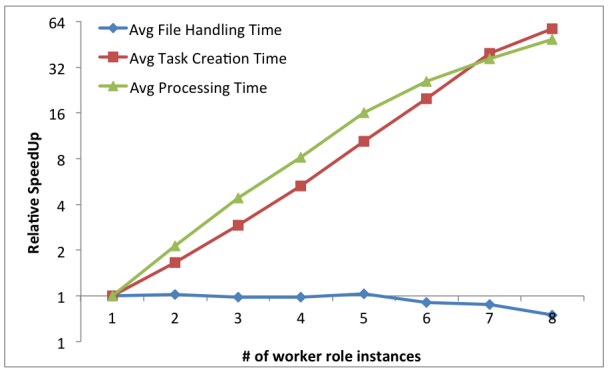
4.5.5 *Crayons* with Larger Data Set

In order to check the effect of data size on the execution time of Crayons we have tried executing distributed dynamic load balancing version of *Crayons* (best end-to-end performer) with the second data set discussed in Section 4.5.3. Figure 4.13 shows how *Crayons* behaves with larger data sets. Since, the load is comparatively uniform, as shown in Figure 4.5, *Crayons* shows much better performance for this data set. The relative end-to-end speedup of *Crayons* goes to more than 40x. The speedup for subprocess of task processing is more than 48x, and for the subprocess of task creation it is more than 57x as shown in Figure 4.13(b). The file handling time demonstrates the similar behavior as for the smaller data set and thus again verifies our claims on network latency from Section 4.5.4.

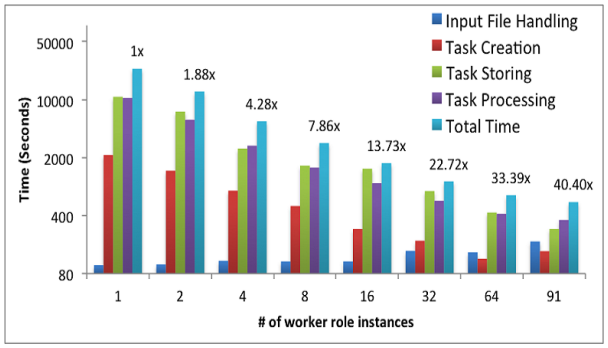


	1	2	4	8	16	32	64	91
Avg File Handling Time	100.52	98.41	102.89	102.28	97.63	110.83	114.21	134.06
Avg Task Creation Time	2143.17	1294.18	736.82	406.63	205.83	108.53	54.61	37.47
Avg Processing Time	10433.08	4867.36	2374.27	1284.27	648.52	405.81	287.62	213.86

(a) Average execution time



(b) Relative speedup



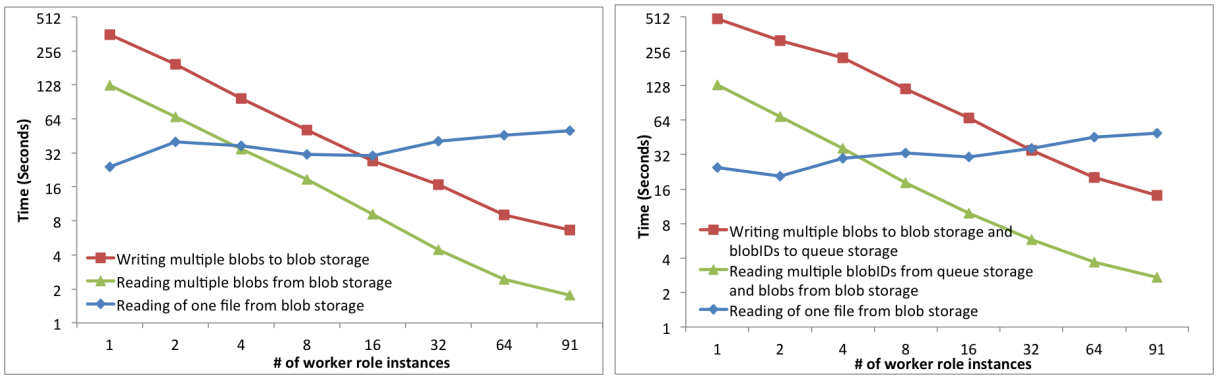
(c) End-to-end execution timings

Figure 4.13 *Crayons*' version with distributed dynamic load balancing using *larger data set*

4.5.6 Scalability of Azure Storage

Azure cloud platform provides various mechanisms to store information for persistent storage purposes and temporarily for communication purposes. *Crayons* utilizes Blob and Queue storage mechanisms. The input files are stored in Blob storage and are downloaded by each worker simultaneously. In the next phase, independent tasks are created and stored in Blob storage. For distributed dynamic load balanced version, for each created task a pointer is stored in the Queue storage which in turn is downloaded by the worker to identify next task to process. The stored Blobs (tasks) are read by worker roles and processed.

Figure 4.14 shows the behavior of Azure's storage mechanism with increasing number of worker role instances. It can be seen that the time for downloading file from Blob storage tends to increase with increasing number of worker role instances. Interestingly, in an earlier version of *Crayons* with previous APIs this phase was a sever bottleneck. The release of



(a) Reading and writing to Blob Storage (b) Reading and writing to Blob and Queue Storage

Figure 4.14 Azure's Blob and Queue Storage Mechanisms' Scalability

newer APIs has addressed this problem to certain extent.

The curve for writing to storage combines the time taken to store the tasks into Blob storage as well as the time taken to write task IDs into task pool (for distributed dynamic version only). This phase scales well, with a speedup of about 61x using 91 worker roles for distributed static load balanced version, indicating the capabilities of Azure platform to support parallel writing to Blob storage. Interestingly, for distributed dynamic load balanced version, the writing phase demonstrates a maximum speedup of 34x using 91 workers pointing to the fact that Azure does not support parallel writing to Queue storage as well compared to parallel writing to Blob storage. Writing phase for distributed dynamic version takes considerably more time compared to the time taken by distributed static version. The extra time is spent in writing to Queue storage (task pool).

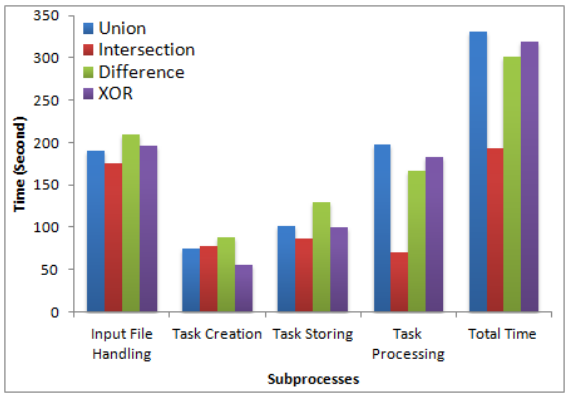
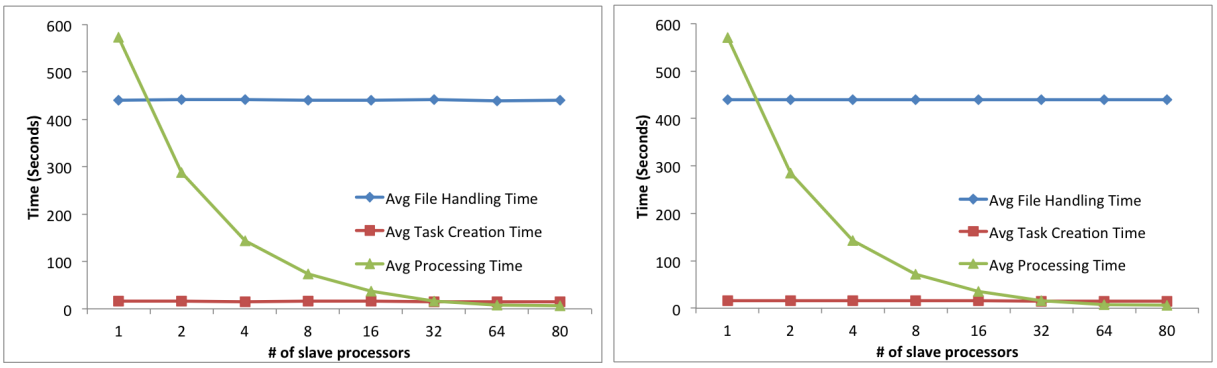


Figure 4.15 Operations supported by *GPC* Clipper library

Similar to the writing phase, reading phase also involves reading from the Blob storage for both versions and reading from the Queue storage, in addition, for distributed dynamic



(a) Static Load Balancing (b) Dynamic Load Balancing

Figure 4.16 Average execution time taken by MPI version of *Crayons*

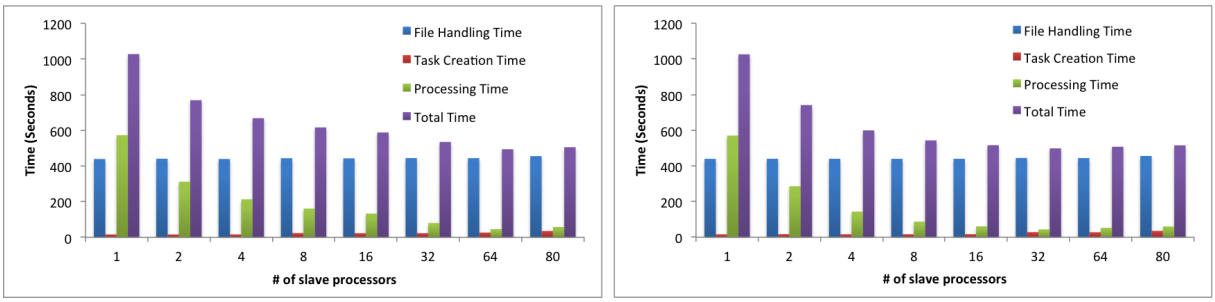
load balanced version only. Interestingly, the difference in timing between these two versions is not as prominent as it was for the writing phase. Logically, it can be concluded that Azure Queue storage mechanism supports parallel reading better than parallel writing as one would expect.

4.5.7 Other Clipper Operations

As discussed in Section 4.2.4, Clipper library supports four operations - Intersection, Union, Difference, and exclusive-or (X-Or). We have utilized the best performing version of *Crayons* (version with distributed dynamic load balancing) using 91 workers to perform all these operations.

Figure 4.15 shows the individual timing characteristics of input file handling, task creation, task storing, task processing phases and the total end-to-end execution time taken for each operation. In order to decide which polygon operates on which polygon from other layer, we use Algorithm 6 to create the intersection graph.

Analyzing the task processing curves, it can be seen that intersection operation is the least compute-intensive operation. All of the rest three operations are almost equally computationally intensive.



(a) Static Load Balancing (b) Dynamic Load Balancing

Figure 4.17 End-to-end execution timings for subprocess of MPI version of *Crayons*

4.5.8 *Crayons* using MPI on Linux Cluster

We have also ported two versions of *Crayons* using MPI on a linux cluster with 80 cores[64]. The cores are distributed among 5 compute nodes. The cluster contains (i) four nodes with each having two AMD Quad Core Opteron model 2376 (2.3 GHz), (ii) one node with four AMD Quad Core Opteron model 8350 (2.0 GHz), and (iii) four nodes with each having two Intel Xeon Quad Core 5410 (2.33 GHz). In our Linux cluster all the nodes share same file system hosted at the head node. Instead of web role/worker role model, here we use master-slave model. For detecting intersection between polygons, we use spatial index structure known as an R-Tree which works well with non-uniform data [65]. First of all, we create an R-Tree by inserting bounding boxes of all the overlay layer polygons. Then, for each base layer polygon, we find the potentially intersecting overlay layer polygons and create intersection graph by linking the base layer polygon to the corresponding overlay layer polygons.

In the first version, we use static load balancing and in the second version we use dynamic load balancing. In both versions, each slave process is responsible for downloading and parsing input files, creating R-Tree indices, and computing polygon overlay. This is the reason why the average timings, as shown in Figure 4.16, for both versions are almost identical.

The master process is responsible for detecting the termination of overlay processing

by slave processes. Once all slave processes terminate, master process merges all the output files into a single output file.

Figure 4.17 shows the execution time breakdown of the static version. Task creation step includes creation of R-Tree, searching for polygon intersection and creation of intersection graph which is used for partitioning the data among slave processes. Overlay processing step includes writing the local output polygons to GML files. In the second version, we use dynamic load-balancing where master process dynamically sends the start and end index of a portion of base layer polygons to slave processes for overlay processing. The reported subprocess timings represent the time taken from first starting instance of that subprocess at any worker to the last finishing instance of that subprocess at any worker.

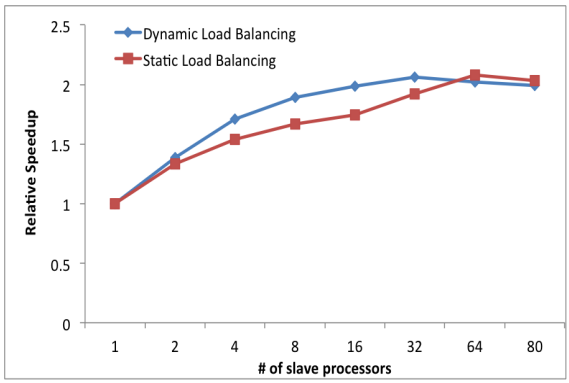


Figure 4.18 Relative Speedup

We experimented with different grain sizes in order to maximize the speedup. Since the size of the message is very small, the communication cost is small in comparison with the cost of input file parsing and data partitioning. As the number of slave processes increase, the average overlay processing time decreases as shown in Figure 4.16.

Figure 4.18 illustrates the relative speedup of *Crayons* system for static and dynamic versions. Again, the file handling remains the bottleneck and results in very poor speedup. Excluding the time taken for converting GML files to *GPC* library's proprietary input format, the speedup for this version was more than 7x. The sequential file handling is a bottleneck and has plagued the end-to-end relative speedup.

4.6 Conclusion

In this chapter we have documented the details of an open-architecture-based overlay-processing system that addresses the critical issues, that have hindered the research for an

important class of data intensive irregular applications over Azure cloud platform. Our results are very promising showing 10 to 40 fold speedup for end-to-end processing. We have initiated collaboration with GIS and Health Policy researchers to employ Crayons for domain science applications. The detailed experimental analysis points to current bottlenecks for strong scaling of skewed, small data sets, and provides roadmap for further research. The system enables experimenting with third party overlay solutions for fundamental GIS operations based on user preferences. For complex GIS inputs (e.g., a hurricane swath over a huge terrain, or unusually skewed or smaller workload), load imbalance among worker roles may be severe. Therefore, it is safe to conclude that *Crayons*' version with distributed dynamic load balancing is superior to other two versions.

PART 5

AZUREBOT: A FRAMEWORK FOR BAG-OF-TASKS APPLICATIONS

Windows Azure is an emerging cloud platform that provides application developers with APIs to write scientific and commercial applications. However, the steep learning curve to understand the unique architecture of the cloud platforms in general and continuously changing Azure APIs specifically, make it difficult for the application developers to write cloud based applications. During our extensive experience with Azure cloud platform over the past few years, we have identified the need of a framework to abstract the complexities of working with the Azure cloud platform. Such a framework is essential for adoption of cloud technologies. Therefore, we have created AzureBOT—a framework for the Azure cloud platform to write bag-of-tasks class of distributed applications. AzureBOT provides a straightforward and general interface that permits developers to concentrate on their application logic rather than cloud interaction. While we have implemented AzureBOT on Azure cloud platform, our framework design is generic to most of the cloud platforms. In this chapter, we present the detailed design of our framework's internal architecture, the APIs in brief, and the usability of our framework. We also discuss the implementation of two different applications and their scalability results over 100 Azure worker processors. Cloud computing is a promising computing resource for both commercial and scientific applications, thanks to the large scale storage capabilities, ease of access by both advanced and novice users, on demand availability, zero maintenance infrastructure, sustainability, and portability of the cloud platforms. The utility based framework facilitates experimenting with large amount of compute power while obviating the need to own a parallel or distributed system[3]. Large scale applications with varying load cycles are perfect fit for the pay-as-you-go cost model of cloud computing as the resources in cloud platform can be allocated and deallocated on-demand based on the application's requirements.

There have been some initial work to understand the benefits and drawbacks of this new framework [4, 11, 12] of cloud computing. Although cloud computing has many aspects closely similar to the traditional parallel and distributed computing platforms, it poses a new set of its own challenges. Traditional large-scale computing resources were not targeted at enabling end-users to rent compute hours with provisioning time being in minutes. On the contrary, cloud computing facilitates anyone to experiment with an idea on a massive platform without investing the capital in owning the resources first, thereby it has the potential to target a much bigger set of users not necessarily familiar with the parallel or distributed computing aspects. Moreover, in order to enable the HPC researchers who currently work with large distributed computing systems, but do not work with cloud computing, it is essential to provide them with easier means of applying their knowledge to cloud computing to bring their expertise to cloud computing. Since cloud computing is still emerging, there is a lack of good programming tools to accelerate the application development process due to inherent complexities posed by the unique architecture of the cloud platforms.

Windows Azure is an emerging commercial cloud platform developed and operated by Microsoft [66]. It provides the developers with APIs to write applications for its cloud services. However, the steep learning curve to learn how to write applications for cloud platform is a big challenge. In this chapter we are presenting our framework AzureBOT that helps applications developers, both professional and researchers alike, to increase their productivity when writing applications for cloud platforms.

Our own experience over the past few years with the Windows Azure cloud platform[67–69] gave us insights into the challenges posed by this emerging platform. Learning from these lessons we have created a framework AzureBOT, that abstracts the complexities of the Azure cloud platform to provide developers and researchers with a simple, user friendly, and robust framework to quickly write applications for the Azure cloud platform. Although our framework is implemented on Azure, the design and internal architecture is portable to other cloud platforms such as Amazon's EC2. However, the primary motive behind designing AzureBOT is not to write a middleware that facilitates porting of applications,

but to abstract the complexities of writing bag-of-tasks applications for cloud platforms and its current implementation does so for the Azure cloud platform.

Our specific technical contributions are as follows:

1. Design of the AzureBOT framework for accelerated application development on cloud platforms for bag-of-tasks applications.

2. Implementation of the AzureBOT framework on the Windows Azure cloud platform.

3. Making an intelligent on the fly choice between Azure Queue storage service and Azure Blob storage service for data transfer at individual task level.

4. Proving the efficiency of our framework by way of implementation of two different applications over 100 Azure processors. We demonstrate the performance and results of AzureBOT for both fine-grained and coarse-grained applications.

To the best of our knowledge AzureBOT is the first framework that allows writing of bag-of-tasks applications on the Windows Azure cloud platform very quickly, without even interacting with any of the Windows Azure storage APIs directly. The users are given access to the simple APIs, which in turn make the decisions for the users to select the best performing storage service at a fine-grained level.

We have implemented two different applications using AzureBOT, an application to scrape data from the world wide web that resembles many of the data crunching commercial applications and a bag-of-tasks application where each task can have a randomly selected processing time to process each task. We also demonstrate the performance of AzureBOT for both of these applications under varying degree of task grain as well as under varying scale of underlying architecture.

Rest of the chapter is organized as follows. Section 5.1 reviews the literature for related work and introduces the Windows Azure cloud platform. Section 5.2 details the design of our framework and the implementations of the two applications using AzureBOT is presented in Section 5.3. Results of our experiments are presented in Section 5.4 and finally Section 5.5 concludes this chapter.

5.1 Related Work

In order to motivate application developers to write robust parallel applications, parallel and distributed community has produced a large number of frameworks and programming models. Some of the popular frameworks are PVM [23], Quincy [70], Map-Reduce [30], message passing interface (MPI) [71], bulk synchronous parallel (BSP) [72], and Dryad [73] among others. Frameworks for cloud platforms exclusively have also been proposed recently such as Twister [74], framework by Bonakdarpour et al. [75], and Publish/Subscribe (Pub-/Sub) middlewares [76, 77]. The key contribution of these frameworks is that they make it easier for a software developer to write softwares for large-scale parallel and distributed computing systems such as the compute-clusters, grid computing, and cloud platforms.

The challenges in working with cloud platforms have recently started drawing interest of researchers and there are a few solutions that offer specific solutions to these challenges. Ekanayake et al. [77] implemented a runtime to facilitate pub/sub pattern of communication to tackle the problem of communication and synchronization between processes running on Azure cloud platforms. Authors highlight the fact that the durable queues, commonly available in cloud platforms, are not sufficient for applications that send arbitrary sized messages. Gunarathne et al.[78] compare the performance of different cloud service providers for pleasingly parallel (embarrassingly parallel) applications.

Projects such as Apache Whirr[20] and jClouds[21] offer cloud-neutral APIs that allow users to write portable codes by using their APIs. However, the biggest challenge for adoption of cloud computing is the initial learning curve to get started with cloud computing, which is not solved by these APIs. Since the users still need to understand the unique architecture and at least one set of APIs, these projects are useful for developers who are familiar with at least one cloud platform and now either want to move on to another cloud or want to write portable applications.

Closest work to ours is by Mocanu et al. [79] where the authors present a similar framework using Jini and Javaspaces technologies. However, the framework proposed by

authors has a lot of overhead to make sure that the framework is fault-tolerant and also to enable synchronization. AzureBOT exploits the Azure storage services to implement fault tolerance for both master and slave processes.

Our review of the literature points to the fact that a large section of the academic community debates if cloud computing is a new paradigm [3, 8, 10] or just grid computing in a new wrapper. Buyya et al.[10] argue that the cloud computing appears to be similar to the grid computing only at a cursory glance but a closer observation presents a different case. Armburst et al.[3] support the claim of Buyya et al. adding that the cloud computing platform uniquely provides an illusion of infinite available resources. Lee [4] advocates the difference by employing the case of hurricane Katrina in 2005 to conclude that the only answer to the *scientific and operational grand challenge problem* is enormous computer power. However, it is not economically possible to dedicate the required amount of resources for this single purpose. Therefore 1. resources must be shared but available on-demand, 2. the platform should be scalable on-demand, and 3. resources should be easily accessible in a user friendly way over the web.

Foster et. al[8] present a comprehensive comparison of the grid computing and the cloud computing platforms. The authors recognize the similarity in the two platforms in terms of the vision and challenges, but the authors also make a solid case to differentiate the two platforms in terms of scale of operation. The authors agree that the more massive scale being offered by the cloud computing platform can demand fundamentally different approaches to tackle a gamut of problems.

We believe that such confusion has hampered the curious nature of researchers to explore cloud computing. With a vague assumption that there are no challenges that have not been previously posed by various distributed computing platforms such as compute-clusters and grid computing in addition to the amount of efforts required to work with cloud platforms, many of the HPC researchers have not been motivated enough to explore the newer research challenges and opportunities in offering computing as a utility. Therefore, we believe that it is timely for a framework such as AzureBOT to help the adoption of cloud computing by

masses.

5.2 Design of AzureBOT

Figure 5.1 shows the typical setup of a bag-of-tasks application that employs AzureBOT framework. The producer and consumer entities are shown to make it easier to understand the general work flow. The arrows that connect with either producer or consumer, e.g. Add new Task, Fetch Task, Get Status, etc., represent the *methods* of the framework that are exposed to the application developers as AzureBOT APIs. The arrows that do not connect with either of the producer of consumers, e.g. Fetch Task Data, Store Task Data etc., are those *methods* of the framework that are not exposed to the application developers.

5.2.1 Implementation

The AzureBOT framework is based on the object-oriented principles, it consists of four classes arranged based on the number of primary real-world entities, which includes 1. the Task class that encapsulates the input data and information about the work to be done, 2. the Result class to hold the output of the processing, 3. the BagOfTasks class to hold the tasks stored in the system, and 4. the TaskExec class that interacts with the processing elements of the system to provide available work and to get results.

Task class The first class is the *Task* class that defines a task. In order to keep it simple, instances of the *Task* class only have an id and a data member. The id can be specified by the end user if the application logic mandates it or it will be auto generated by the system. The data member should be textual in nature. Although this sounds impractical, thanks to the serialization libraries available in .Net framework, any type of data can be represented as text and hence can be stored in the *Task* class.

Result Class The next class is *Result* class which holds the objects that contain the result(s) of input processing. Similar to the *Task* class above, every instance of the *Result*

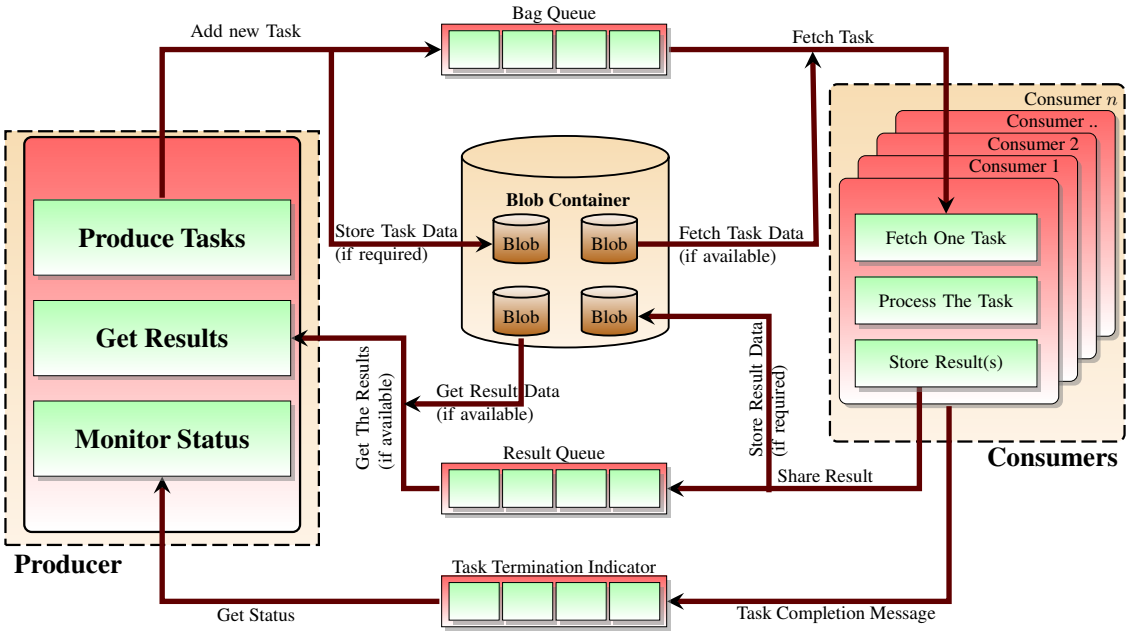


Figure 5.1 A typical setup for a bag-of-tasks application employing AzureBOT framework. The choice of storage service to store data is made on the runtime based on the data size to improve efficiency.

class holds an id and a member to hold the result. The data member that holds the results in this case also can only hold textual data similar to *Task* class above.

BagOfTasks class The next class is the *BagOfTasks* class, which does most of the work for an application. When an object of this class in instantiated, it creates an Azure Queue that serves as the bag to hold the tasks as well as the blob container that might be required to store the large data for tasks and results. The constructor requires the bagName to be passed, and the bagName should be unique.

Moreover, a variation of the default constructor accepts a list of workers, if a bag is to be assigned to a subset of available workers. There can be multiple bags existing at the same time, but one worker can only be assigned to one bag. One queue is created for every worker involved in the process and the name of the bag is stored as a message in these individual queues. In order to avoid clutter, we have not shown these individual private queues in Figure 5.1.

BagOfTasks class exposes following APIs to the user:

addTask(Task): This API takes an object of the *Task* class as an input and adds it to the bag. The framework at this point determines if the data size of the task is small enough to fit in the queue. If it can fit in a queue the task is stored in the queue itself, otherwise a new blob is created to hold the data and the blob-id (name of the blob) is stored in the bag queue. There is also a static version of this API which allows adding a task to the bag without initiating an instance of *bagOfTasks* class. The static version allows an application to have multiple producers adding tasks to the same bag. The limit on task size is controlled by the scalability limitations of the Azure Queue storage, and it currently happens to be 64KB.

getStatus(inOutRatio): This API returns the percentage of work finished by the workers. It takes the number of results produced per data item as input and calculates the progress based on that. The input parameter inOutRatio extends our framework to work on different variations of the bag-of-tasks applications[79]. Based on the inOutRatio parameter, the framework can learn the number of results to expect for each input task to calculate progress according to that.

The number of processed tasks is the number of messages in the Task Termination Indicator queue. There is another queue which is used to store the number of input tasks, this queue is not shown in Figure 5.1 to avoid cluttering the diagram as this queue serves only a single purpose.

There is a static version of this API *getStatus(bagName, inOutRatio)* which can be used for long running applications. The user can come back at a later time and query the status of a bag by specifying the bag's name.

getResult(): This API fetches one result from the result queue and returns it to the caller. The result might be stored in the queue itself if it is not larger than that can fit in a queue. Otherwise, the result is fetched from the Blob storage.

In order to facilitate multiple consumers to fetch results there is a static version of this API as well. The static version takes bag's name as extra parameter and does the same thing as the instance member. The static version is useful when the same machine behaves as both

producers and consumers, and multiple steps are required in order to complete the entire work flow of an application. For instance, an application may convert the results of one computational bag-of-tasks overflow and feed them as input to the next step by converting the result objects into task objects, which is simply a matter of copying the result id and data over to the task id and data.

getResultCount(): This is an additional API if the application logic demands to know the number of results instead of the current progress, which is returned by the *getStatus* API. As the name suggests, this API returns the raw count of messages in the Result queue. Following the practice from the *getStatus* API, there is a static version of this API as well.

deleteBag: This API is used to delete the data related to the bag stored in the Azure storage account. The static version of this API takes bag's name as input. All queues, blobs, and blob containers are deleted as a result of this API's execution. The deletion commands check if the object is present before deleting, thus the API is safe from multiple consumers calling deleteBag at the same time.

TaskExec class The last class is the *TaskExec* class, used by the workers (consumers) to fetch and execute a task and to store the results. When an instance of this class is created by a consumer, it finds out which bag is assigned to this worker or which bag this worker is assigned to. We currently limit only one bag to be assigned to one consumer but as we develop more applications using AzureBOT framework, we will learn if we should allow more than one bags to be assigned to a consumer.

The *TaskExec* class has the following APIs:

fetchTask(): This API checks if there are tasks in the bag queue and, if found, returns one task to the consumer. It also keeps track of the task given to the current consumer. If the consumer fails during the processing, the task is put back in the bag for the next idle consumer to consume it while the failed consumer restarts. In order to allow this fault tolerance, we set a timeout period for each task. If the worker does not confirm the successful processing of the fetched task within that time it is assumed that the consumer has stopped

working and the task reappears in the bag automatically. Moreover, a reference to the task is stored within the instance so that when a worker reports successful completion the referenced task can be deleted.

storeResult(Result): This API takes an object of *Result* class and stores it in the result queue. The AzureBOT framework checks if the result can be stored in the queue entirely or if it requires a blob. If a blob is required, then a blob is created and the data is stored in the blob instead of the queue. However, to inform the result queue of the location of the result data, blob's id is stored in the result queue.

signalEndTaskProcessing(): This API helps the AzureBOT framework to keep track of the current progress of the status. After every successful task completion the consumer is supposed to execute this API which in turn will add a message to the Task Termination Indicator queue. In addition to adding a message to the Task Termination Indicator queue, this API also deletes the task from the bag permanently. The APIs *getStatus(inOutRatio)* and *getResultCount()* from the *BagOfTasks* class depend on the Task Termination Indicator queue to get the number of finished tasks.

5.2.2 Limitations of AzureBOT

Although AzureBOT is first of its kind, we have identified a few limitations that make it unsuitable for certain applications with special requirements. First and foremost, if the application logic requires explicit synchronization there is no direct way of achieving it using current version of AzureBOT. There are however workarounds such as creating a new bag to replace the synchronization in the application logic by feeding the results of the previous processing step as tasks to the next bag to be created. However, this only affects a small number of bag-of-tasks applications as by definition the tasks in a bag-of-tasks application should be independent.

Secondly, current version of AzureBOT is written using C# programming language and hence is only usable for other applications written in C#. We are planning to release a version of AzureBOT in future that would support multiple languages that are supported

by the Azure cloud platform such as PHP.

Finally, AzureBOT is only for those applications written in C# that are hosted in the Azure cloud platform. There are applications (written in C#) that run on other platforms, or even locally, and offload tasks to Azure cloud platform for which AzureBOT does not fit well.

5.3 Implementing Applications using AzureBOT

We have successfully implemented two different applications using AzureBOT. In this section we will talk about the applications, the development time spent in writing these applications, and the efforts required to implement these applications. We do not use any of the APIs from the Azure SDK directly, but the applications use AzureBOT APIs. The number of lines of code is less than 300 lines for both applications combined. The implementation time is an impressive less than 5 hours for both applications, which includes testing and debugging.

Our code for both applications can be found at `http://www.cs.gsu.edu/dimos/?q=content/azurebot.html`; please note that the published code is little different than the original code as the profiling statements have been taken out.

5.3.1 Internet Data Scraper

Internet data scraper application behaves similar to any search engine, only that it does not calculate the rank of the pages but just scrapes them. The application takes a list of URLs and adds each URL as a task to the bag. The consumers (workers) fetch one task at a time, extract the URL from it, read the HTML on that page, and finally collect all the URLs (links) on those pages. These URLs are stored as a large text data delimited by a special character not allowed in URLs and then this text data is stored in a *Result* object which is stored in the Result queue.

While the task data will fit in the queue storage for most of the normal sized websites, for larger sites with multiple URLs per page the result data does not fit in the queue. In this

case, AzureBOT was able to handle this by creating blobs. Since the performance takes a hit by adding an extra step of storing data in the blob storage, our first preference always is to store data in the queue, unless of course the data is too big to be transferred as a message in the queue.

The producer end of this application has less than 100 lines of code and the consumer end of this application took less than 50 lines of code. It is worth mentioning that we pass the HTML read to a third party HTML parser to get those HTML elements that represent a link on the page.

The total time to code the entire application was around 2 hours and additional 1 hour was spent in testing and debugging, most of which was due to the learning curve for the third party HTML parser library.

5.3.2 Master Slave Simulator

Master slave simulator is an application that simulates the behavior of a typical master-slave application. The Master (producer) creates a task with parameterized random thinkTime stored as the data (instructions) for the task where thinkTime represent the time it would take a real world application to process the given data for the respective task. For instance, in the case of the Internet data scraper application thinkTime would be the time it takes to fetch the page hosted at a given URL and to parse the HTML to get all of the links from it. The slaves (consumers) fetch a task and think on it for the amount of time specified in the task data. Then the slaves record the time took to process the task and store it in the result data. The results are then added to the result queue from where the master gets them.

The total lines of code for the producer end for this application is also less than 100 and for the consumer end it is 25 lines. The code took about 1 hour and the testing and debugging processes took less than 30 minutes.

5.4 Performance and Results

In order to test AzureBOT framework with varying number of worker role processes, we have tried running both applications described in Section 5.3 using up to 100 total processors. For both applications we keep 1 web role and vary the number of worker roles from 1 through 99. The role instances are configured as small role instances on Azure cloud, which means each role is a single core virtual machine with 1.75 GB of memory, around 20 GB of disk space (not used in our case), and 5 Mbps of allocated bandwidth.

The ratio of input tasks to results (inOutRatio) was set to the default value of 1 for both of the applications. We have also ignored the variations in results that could be attributed to the time of the day and the day of the week. Cloud platforms typically exhibit different behaviors at different time periods because of their multi-tenant environment. However, we did not see any major variation or significantly different values than what we expected.

5.4.1 Internet Data Scraper

For the Internet data scraper we prepared a list of approximately 1400 URLs that were fed as input. This list had a mix of smaller and larger web pages with varying number of internal and external links, and hence a large number of results were stored in blobs and rest were communicated over a queue.

Figure 5.2 shows the results of this experiment. The download speed of the Internet connection used to fetch the pages is about 100 Mbps and the upload speed is about 40 Mbps (averaged over 10 tests), and most of the tasks, out of 1400 total, took less than 1 second for getting the content of the page, processing it, and storing the results in the bag. However, there were tasks taking as much as 10 seconds to process.

It can be seen that the processing time on the master end stays the same as we always use 1 role instance for master end. The processing time for workers initially shows a good speedup but quickly saturates as the number of instances grows beyond 4. The reason as can be easily guessed is that the producer is not able to supply enough work for the workers.

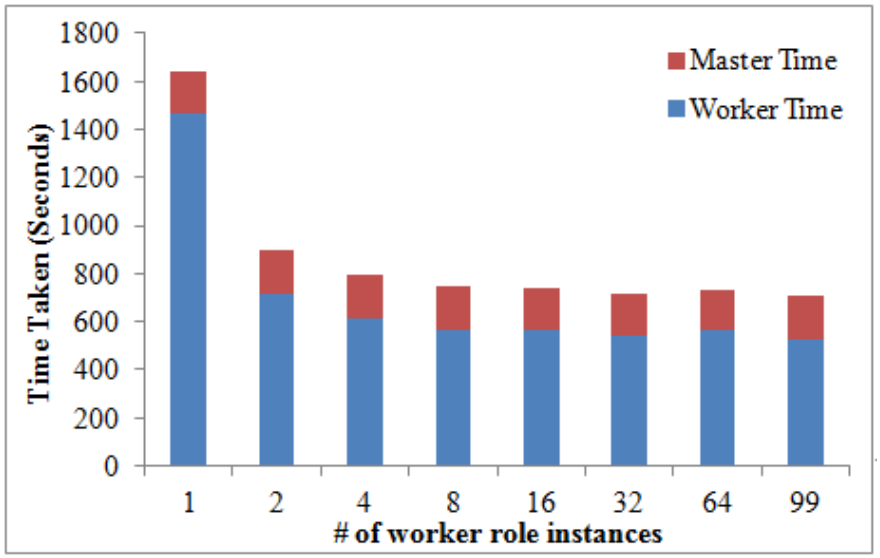


Figure 5.2 Performance of Internet data scraper application over varying number of Azure worker role instances.

For large number of workers, the time starts going up as the idle workers are creating overheads by querying the bag for messages. We have controlled this overhead to a greater extent by designing AzureBOT in such a way that if there is no task for a worker, the worker will only be able to query the bag again after 1 second. This delay also helps AzureBOT to avoid throttling the Queue storage with the capacity of handling 2000 requests per second.

It should be noted that AzureBOT can be used to write applications where workers themselves act as both producers and consumers. In this case the only requirement would be that the input should be divisible into multiple independent data sets.

5.4.2 Master Salve Simulator

Master slave simulator application allowed us to tweak the grain of the task to test AzureBOT against fine-grained to coarse-grained applications. It provided us the flexibility to hold for one variable and to test another. We have analyzed the effect of varying number of worker role instances while the thinkTime (grain of the application) is constant as well as varying the thinkTime while keeping the number of instances constant.

Figure 5.3 demonstrates the performance of AzureBOT with varying number of worker

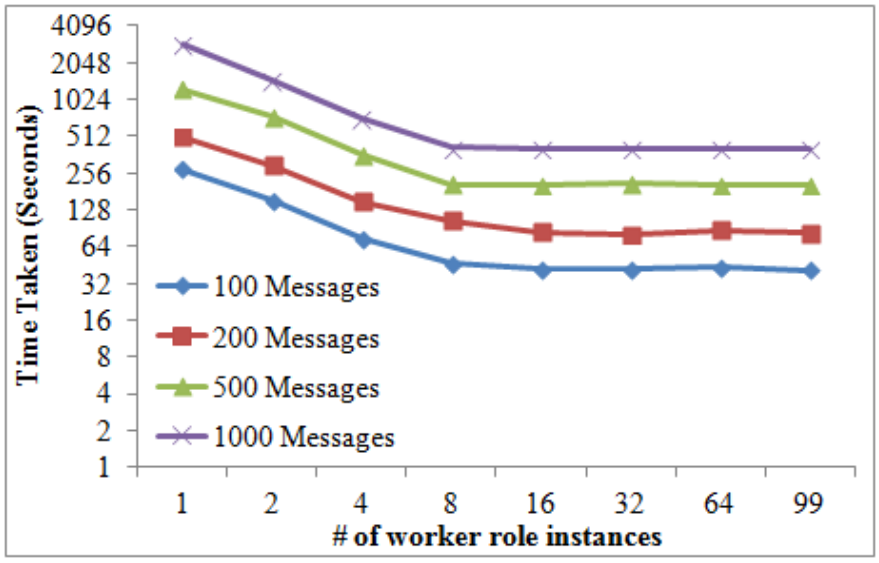


Figure 5.3 AzureBOT performance over varying number of worker role instances; max thinkTime = 5 seconds.

role instances over different number of messages (tasks). The Number of messages define the amount of work available and hence higher the number, the more avenues for workers to be productive. The maximum thinkTime was set to 5000 miliseconds so the application would best qualify as a fine-grained application.

The results are again as expected. Similar to the Internet data scraper application, the demand supply imbalance is playing a key role here too. The time taken to process a task initially reduces with the increasing number of worker role instances, but quickly saturates as the number of worker role instances reaches 16. The observation holds for the case of as few as 100 messages in the system and as many as 1000 messages in the system, which again suggests that the amount of work is not the limiting factor but the imbalance between the messages that can be consumed and the number of produced messages.

The time on the Master end stays constant throughout the experiment as regardless of the number of worker role instances, the master does the same processing every single time.

Figure 5.4 shows the effect of increasing application grain when the number of messages and the number of role instances are kept constant. Interestingly, the processing time is almost the same till the maximum thinkTime of 10 seconds and then it starts growing

proportionally. Again we can see that for smaller grains the workers finish their job quickly and go back to the bag for more work, and hence the producer fails to keep up with the demand. While for large thinkTime values the producer can produce more than enough work for all available workers before a worker can consume a single task and hence the effect of thinkTime becomes more prominent.

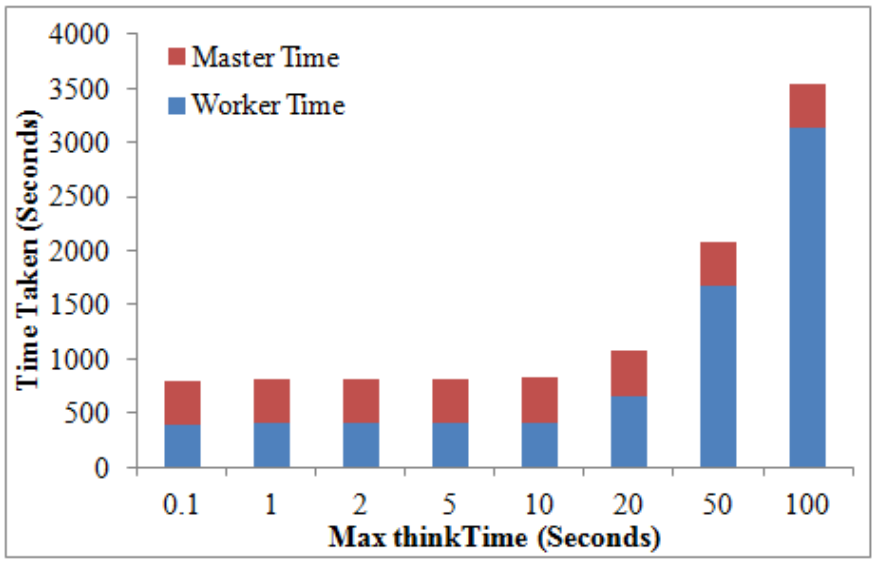


Figure 5.4 AzureBOT performance over varying application grain; number of worker role instances = 16 and number of messages = 1000.

In order to assess our AzureBOT framework we also implemented the master slave simulator without using AzureBOT framework. Since we used some of the existing libraries from AzureBOT framework for this implementation, time to code is not a good indicator for this comparison. Lines of code, however, clarify the difference. The total lines of code without AzureBOT is over 500 lines. We further note that for master slave simulator the messages are not too large and hence we could avoid dealing with the blob storage, thereby obviating the need to employ the libraries from AzureBOT framework that communicate with Blob storage. Otherwise, the code size would have grown twice in size.

Figure 5.5 demonstrates the performance of master slave application without AzureBOT framework. It can be seen that the performance is almost the same. The minor time difference could be attributed to the fact that cloud platform is a multi-tenant environment.

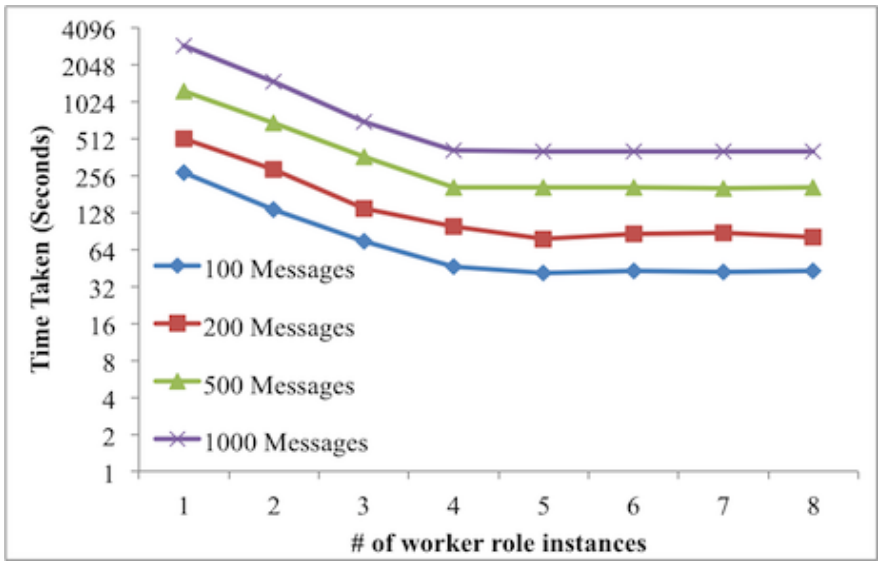


Figure 5.5 Performance of version without AzureBOT, over varying number of worker role instances; max thinkTime = 5 seconds.

Unlike traditional Linux cluster where the developers could choose and schedule machines in a controlled environment, it is out of application developers hands to select the machines where the code is going to execute.

5.5 Conclusion

In this chapter we have introduced AzureBOT framework for accelerated application writing for cloud platforms. Our results and development times are a testimony to the fact that the AzureBOT framework is highly effective at increasing the productivity of the application developers and thus can greatly help with adoption of cloud computing at large scale. With the meager amount of efforts required to port bag-of-tasks applications to cloud platform, we believe that the researchers will be motivated to explore cloud platforms.

AzureBOT currently has a few limitations that we are going to address in the future version; we have created AzureBOT for applications hosted on the Azure cloud platform. If a developer wants to use AzureBOT framework for a desktop based tool, e.g. the producer is a desktop based application that sends work to the cloud based workers, by adding tasks to a bag hosted on cloud, it is currently not possible. However, this limitation does not affect

the functionality of the AzureBOT framework currently.

PART 6

CLOUDMPI - A FRAMEWORK FOR MPI-STYLE APPLICATION DEVELOPMENT ON THE AZURE CLOUD PLATFORM

Cloud computing platforms are getting popular by the day among small and large business enterprises who view cloud computing as a service that is easy to use, available on demand, has zero maintenance infrastructure, and supports sustainability and portability. However, the parallel and distributed community, both academia and industry, still does not perceive of cloud platform as a resource that can replace traditional computing platforms. Furthermore, the advanced parallel programming skills acquired over a period of time are not easily applicable to the architecture of the cloud platforms. Although cloud vendors that offer IaaS (Infrastructure as a service) allow customized software installations to mimic the capabilities of a virtualized compute cluster node, IaaS cannot demonstrate the network capabilities of a traditional cluster.

Cloud platforms, however, have the potential to be the resource of choice for high performance computing (HPC) community. Literature reports various studies comparing the pros and cons of cloud platforms [4, 11, 12]. Even though there is a large number of big and small cloud vendors available in the market today, the service offerings have little similarity. Recent trends show that the cloud vendors have realized the need of such uniformity [20, 21, 80–82] to maintain the competitive edge. However, the current state-of-the-art is still far from ideal, especially from the end-users' perspective.

The steep learning curve involved in understanding the very peculiar and non-uniform architectures and runtime environments of various cloud platforms discourage HPC community to adopt it as the platform of choice. In order for a large-scale penetration of cloud computing platforms into the HPC community, cloud vendors will have to offer easier approaches to utilize these platforms for scientific research. Our experience over the past few

years with the cloud platforms suggests that the best approach to help HPC community to bring their expertise to cloud computing is by providing them with such runtime environments and tools that they are familiar with from traditional parallel and distributed setting.

One of the popular set of standards followed by the HPC community is the message passing interface (MPI). Various implementations of MPI standard are available in the market proving the effectiveness of MPI standards for parallel and distributed application development. While MPI has been a popular choice for traditional parallel and distributed platforms, its current implementations are not pragmatic for cloud computing platforms[83]. Literature reports that the performance of an application when ported to cloud using state-of-the-art MPI implementations was as much as 35x worse than the same setup on a compute cluster[31].

Windows Azure cloud platform is an emerging commercial offering from Microsoft [66]. Our previous experience with the Azure platform developing large GIS applications and benchmarking Azure's storage services[67, 69] gave us insights into performance bottlenecks of the Azure cloud platform. The primary reason for such bad performance of MPI implementations on the bare-bone cloud infrastructure is the dependency on the network interface card, which is shared among multiple tenants in the cloud environment. Moreover, we suspect that bypassing the vendor specific APIs for communication leads to a competition for the message buffer space and thus the performance degrades further.

Our thesis is that to gain best performance and to maintain programmer's productivity, one should apply separation of concerns; that is, exploit cloud APIs which increasingly will become sophisticated to deliver performance of cloud black-boxes and provide MPI-like interface most familiar to HPC developers. With this as our guiding principle, we have created a framework called cloudMPI. Our framework abstracts the intricacies of the Windows Azure cloud platform and allows developers to write application as if they were writing any traditional MPI application.

To the best of our knowledge, cloudMPI is the first framework that allows writing

MPI-style applications on the cloud platforms without sacrificing the performance of the underlying platform. While our implementation is specific to the Windows Azure cloud platform, our design is applicable to other cloud platforms such as Amazon EC2.

Our specific technical contributions are as follows:

1. The design of the cloudMPI framework for MPI-style application development on cloud platforms. This includes the design of MPI-like communicators and primary communication primitives for send, receive, and broadcast, supplemented by a new primitive, RecvMult(), to receive a batch of messages. We think that the latter can further reinforce the notion of messaging frugality in parallel program design. The underlying cloud implementations such as Azure can efficiently support such batch receives.

2. Efficient implementation of the cloudMPI framework on the Windows Azure cloud platform.

3. Introducing an intelligent on-the-fly choice between Azure Queue storage service and Azure Blob storage service for data transfer at message level based on message size.

4. Proving the practicality and efficiency of our framework by way of porting a complex application, Crayons, from the domain of GIS and comparing the performance, programmer productivity, and lines of code. In our experiments the performance was affected by at most 2%, while we reduced over 90% of the lines of code that were there to interact with cloud storage.

Rest of this chapter is organized as follows. Section 6.1 reviews the literature for related work and introduces the Windows Azure cloud platform. Section 6.2 details the design of our framework along with the APIs. The application Crayons and its porting is presented in Section 6.3. Results of our experiments are presented in Section 6.4 and finally Section 6.5 concludes this chapter with comments on future work.

6.1 Background and Literature

While there are a large number of frameworks such as PVM [23], Quincy [70], Map-Reduce [30], bulk synchronous processing (BSP) [72], and Dryad [73] among others, MPI [71] is by far the most popular set of standards for application developers to write robust parallel applications[25]. There has been a lot of research around these MPI standards and a lot of implementations are available in the market[26–29].

Some exclusive frameworks for cloud platforms have also been proposed recently such as Twister [74], framework by Bonakdarpour et al. [75], and Publish/Subscribe (Pub/Sub) middlewares [76, 77], but every such framework has a particular set of complexities that discourage users from using these frameworks.

There is a steep learning curve for even a veteran eScience developer to understand the idiosyncrasies of cloud platforms besides the APIs to work with one. This has created a number of research opportunities to offer solutions to these challenges. Ekanayake et al. [77] implemented a framework to facilitate pub/sub pattern of communication to facilitate communication and synchronization between processes running on the Windows Azure cloud platform. Authors highlight the fact that the durable queues, commonly available in cloud platforms, are not sufficient for applications that send arbitrary sized messages. Gunarathne et al.[78] compare the performance of different cloud service providers for pleasingly parallel (embarrassingly parallel) applications.

We have also created a framework AzureBOT for bag-of-tasks application development on Azure cloud platform that allows users to quickly write scientific applications with independent tasks[84].

There are also projects that try to attack the problem from a different view. Instead of creating individual solutions for every cloud platform, these projects offer a set of uniform APIs that can underneath connect with any of the multiple cloud vendors. Projects such as Apache Whirr[20] and jClouds[21] are examples of such projects. However, these projects do not solve the problem of steep learning curve to work with the cloud platform. The end

user is expected to have expertise in at least one cloud platform and some familiarity with the rest.

6.2 Design of cloudMPI

6.2.1 Design philosophy

The lack of an MPI implementation for cloud platform is clearly not due to the lack of expertise or interest, but designing the MPI runtime environment for cloud platforms is a non-trivial research problem. Many of the cloud vendors follow Software as a Service (SaaS) model and thus offer their own APIs to access resources in the cloud. The traditional MPI implementations had the luxury of knowing the details of the underlying infrastructure, which is not the case anymore. The cloud nodes are also more susceptible to failures than those in traditional compute-clusters. Moreover, the multi-tenant environment poses further limitations on resource access and sharing.

With current version of cloudMPI, we only support a small set of APIs from the traditional MPI implementations. Routines that can differentiate between blocking and non-blocking calls as well as synchronous and asynchronous communication routines are not implemented separately. For instance, we decided to only have *cMPI_Send* routine to send a message, rather than having multiple routines similar to traditional MPI implementations. Although this might appear as a sever limitation on a cursory glance, a closer look revealed that multiple routines were not needed in this case. We no longer need to depend on a local message buffer to make sure that the message is sent and the buffer is available for next message. We delegate this process to Azure APIs, which in this case is our best bet as the APIs have the knowledge of the underlying architecture.

Moreover, instead of storing a message locally, we decided to store it in the cloud storage. The choice of the cloud storage service (e.g. Blob storage vs. Queue storage) to be used by a particular message is made by cloudMPI at individual message level.

Another significant difference between the traditional MPI implementations and

cloudMPI is the difference between master and slave nodes. In traditional implementations there is only one program running on every node, the determination of master and slave is based on the MPI Rank. However, in case of Azure, there are two types of compute instances, web roles and worker roles. Typically, web role works as the master and worker roles work as slaves. These machines can have multiple instances each, but the code to be deployed on these machines have to be written separately.

Both types of machines have similar IDs starting at 0, and on top of that there are no APIs that can determine the type of the role where a particular code is running. Thus, sending a message to the node with ID 0 does not accurately specify whether the message is meant for the web role or worker role. Therefore, we maintain a separate queue for master node and a parameter is added to the APIs to separate a message designated for a slave node or master.

Since Azure Queue storage does not guarantee FIFO functionality, implementing ordering of messages is difficult. It is possible that a message sent at time t_1 was retrieved after a message sent at time t_2 where $t_2 > t_1$. This further makes it challenging to implement the collective APIs such as *MPI_gather* and *MPI_scatter*.

Finally, the message sharing concept has been traditionally missing from MPI implementations. With cloudMPI we allow nodes to steal work from other processors. This serves two purposes, (i) it allows developers to write adaptive load balancing strategies in their applications, and (ii) if a node fails the messages sent to this node can be collected by another node that can in turn process them.

6.2.2 Implementation

The design of cloudMPI is similar to the design of traditional MPI implementations, but it is tailored to the Azure cloud platform. In order to avoid the congestion at the network interface card, shared among multiple tenants, we do not rely on point-to-point communications. One local queue is created for every node (virtual compute instance) where messages can be sent by any of the nodes. Moreover, if multiple communicators are

Table 6.1 cloudMPI APIs at a glance

cloudMPI API	**Parameters**	**Storage Services Required**	**Related MPI APIs**
cMPI_Init	Communicator Name, IDs of nodes in the communicator	Table storage: To store communicator metadata, Queue Storage: To create required queues (message buffers) to hold messages	MPI_Init
cMPI_Comm_Size	Communicator Name	Table storage	MPI_Comm_Size
cMPI_getRank	None	None	MPI_Comm_rank
cMPI_Send	Message, Communicator Name, and the type of node	Queue Storage	MPI_Send, MPI_ISend
cMPI_Broadcast	Message, Communicator Name, and whether to send to Master	Queue storage and Table storage	MPI_Bcast
cMPI_Recv	Node ID (self by default), Communicator Name, and the type of node	Queue Storage	MPI_Recv, MPI_IRecv
cMPI_RecvMult	Message Count, Node ID (self by default), Communicator Name, and the type of node	Queue Storage	N/A
cMPI_Delete	Communicator name	Queue storage and Table storage	MPI_Finalize

required there will be one queue per node for each communicator. Communicators can be perceived as communication channels in our model. Our benchmarking results[67] show that using multiple queues offer better performance than using a single queue. Moreover, having multiple channels could be used to maintain the order of the messages.

The cloudMPI framework is based on the object-oriented principles, it consists of two classes 1. cMPI_Message and 2. cMPI. The cMPI_Message class packs the data to be transferred in a message as well as the other attributes of a message including the tag recommended by MPI standard. The cMPI class offers the methods to facilitate the point-to-point communication among MPI nodes.

cMPI class: cMPI class exposes following APIs to the user:
cMPI_Init(firstNode, lastNode, cMPI_COMM): This API initializes a channel of communication and assigns a list of nodes to this channel. One queue, dedicated to this channel, is created per node after the initialization. The need to allocate nodes to a channel allows users to broadcast a message to all of the nodes in a communicator. This API also records the size of the communicator.

cMPI_Comm_Size(cMPI_COMM): Similar to the *MPI_Comm_Size* routine, this API returns the total number of cloudMPI processes in the communicator named by the parameter cMPI_COMM.

cMPI_getRank(): This API is similar to the *MPI_Comm_rank* routine, this API returns the rank of the calling cloudMPI process. It is to be noted that unlike the traditional implementation where the processes get a rank from 0 through (number of tasks -1), here the ranks are dependent on the id of the worker role processor making the call. Thus, the parameters firstNode and lastNode supplied during the *Init* API call define the start and end rank of the processes.

cMPI_Send(cMPI_Message, cMPI_COMM, toMaster): The Send API looks a little different than the send API from that from the traditional MPI implementations. The primary reason for this is that to keep it simple for developers we treat the messages sent to the

master differently than the messages sent among worker roles. It is important to note here that since Azure compute services has web roles and worker roles it is natural to think of the web role as the master and the worker roles as slaves. However, it is not mandatory for these roles to act like that. A worker role could easily be made to behave as a master. The parameter toMaster defines if the message is for the master node or for the slave nodes. This distinction is necessary as the typical Azure setup can have more than one compute instances with the same ID and thus ID 0 does not automatically make the node the master node. Moreover, there is no Azure API that could be used to detect if the calling process is a web role or worker role.

cMPI_Broadcast(cMPI_Message, cMPI_COMM, toMaster): The Broadcast API sends the message to all of the processes except the calling process in the cMPI_COMM communicator. The toMaster parameter defines if the message is to be sent to the master or not.

cMPI_Recv(destination, cMPI_COMM, fromMaster): The routine to receive a message is also different than the traditional implementations. Since we do not create a separate queue for all possible combinations, all messages for a cloudMPI process are sent to the only designated queue. Since Queue storage does not allow querying the messages based on source, there could only be one message that can be extracted from the queue. The destination parameter allows the developers to steal work from other processes for better load balancing if needed.

cMPI_RecvMult(count, destination, cMPI_COMM, fromMaster): The Windows Azure cloud platform allows extracting multiple messages in one call. In one such call a maximum of 32 messages can be extracted. In order to exploit this feature we have introduced the RecvMult API to fetch multiple messages from a queue. Other than that the API is similar to the Recv API.

cMPI_Delete(cMPI_COMM: This API deletes the resources occupied by a communicator.

6.2.3 Limitations of cloudMPI

cloudMPI is the first framework to allow writing MPI-style applications for the Azure cloud platform without sacrificing the performance that can be afforded by using the native APIs. Nevertheless, we still have a few limitations that we would like to address in the next versions of this framework. First and foremost, there is no explicit synchronization among cloudMPI processes. Although, one can implement a functionality like this using an extra channel to work as a barrier, we would like to offer an API to do so.

Our *cMPI_Barrier* API is still under implementation and is scheduled for release in the next version. In order to support the synchronization, there will be a separate queue. Every node will put a message in this queue on reaching a barrier routine during runtime execution. The API will return control to the calling process only after every node in the communicator has inserted a message in this queue acting as the barrier.

cloudMPI does not support the order of the messages, which is required by the MPI standard. This limitation is in part due to Azure Queue storage's inability to support FIFO. We are working on alternative storage mechanisms (or a combination of them) to allow ordering of the messages so that a traditional MPI application can be translated into cloudMPI application.

One way to implement the collective MPI routines such as *MPI_gather* and *MPI_scatter* is to have one queue reserved per node per communicator to support collective calls. The nodes in this case will be restricted from reading messages sent to other nodes. We are also experimenting with adding metadata to the message to distinguish between a normal message and a message that is part of a collective communication. This approach allows avoiding creating too many queues that might not be needed for certain applications.

6.3 Implementing applications using cloudMPI

In order to justify the effectiveness of cloudMPI framework, we have implemented a complex application from the Geographic information systems and sciences (GIS) domain. In

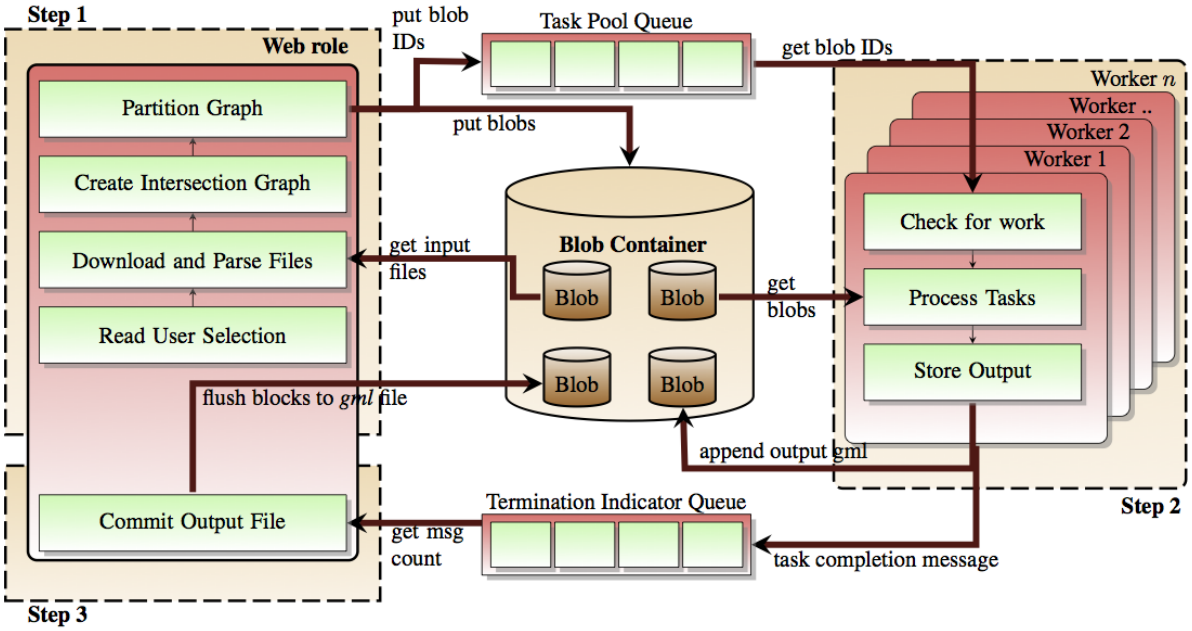


Figure 6.1 Software architecture of Crayons

this section we will discuss the application briefly, the development time spent on porting this application from Azure APIs to cloudMPI, the performance hit taken due to our framework, and the lines of code saved due to this porting.

Our code for the application before porting can be found at `http://gpcoverlay.codeplex.com` and the code after porting can be located at `http://www.gpcmpi.codeplex.com`.

6.3.1 Crayons using native APIs

Crayons is an Azure cloud based application for map overlay process over vector-based (polygonal) GIS data. The polygon overlay process can be visualized as a process where more than one maps are overlaid on top of each other so that the resultant map can be queried for aggregated data. An example of this could be overlaying the map of a city with a map of the swath of a hurricane. The resulting map can be queried to find those rescue centers in the city that are safe.

Figure 6.1 shows the architectural diagram of Crayons with centralized load balanced version employing an extra large virtual machine(VM) as the web role. End users have the option to upload their input files in GML format to cloud or to operate on the existing files. Since uploading is a trivial process we will assume that the files are already available in the cloud storage. User selects the GML files to be processed along with the spatial operation to be performed on these files. First of these two selected files is treated as the base layer and the other file is treated as the overlay layer. The web role immediately starts downloading the files from the Azure cloud storage and translates (parses) the features (polygons) from the input GML files into C# objects.

Since spatial overlay operations are computationally expensive, it is wise to prune the set of polygon pairs needed to be processed together. In order to create this intersection graph, Crayons finds each overlay polygon that can potentially intersect with the given base polygon and only performs spatial operation on these pairs. This is achieved using the coordinates of bounding boxes generated during parsing of input files. Intersection graph creation currently is based on sorting the polygons with $\Omega(n \log n)$ cost.

Intersection graph defines one-to-many relationship between the set of base polygons and overlay polygons. To create an independent task, one polygon from base layer and all related polygons from overlay layer are merged together as a task and stored in the cloud storage as a blob. The web role converts the C#'s polygon objects belonging to a task to their GML representation that gets stored in the blob storage. We prefer in-house serialization against C#'s serialization library to avoid excessive metadata required to convert an object to string.

Each task is given a unique ID, this id is communicated to the worker roles using a message over a queue that serves as a shared task pool among workers and thus facilitates dynamic load balancing. Queue storage mechanism provided by Azure platform comes handy here to implement task based parallelism and for fault tolerance.

Worker roles continuously check the shared task pool (queue) for new tasks. Since this can throttle the Queue storage, with a limit to support a maximum of 500 requests

per second, if there is no message in the Queue we let a worker sleep for a few seconds before sending next request. However, if there is a task (message) in the shared task pool, the worker reads the message and consequently hides it from other workers, downloads the blob with id stored in this message, converts the content of the downloaded blob to get the original base and overlay polygon objects back (deserialization), and performs the spatial overlay operation by passing a pair of base polygon and one overlay polygon at a time to *GPC* library for sequential processing.

GPC library returns the resultant feature as a C# polygon object that is converted to its equivalent GML representation and appended as a block to the resultant blob stored in the cloud storage. Azure API *PutBlock* is used to achieve parallel writing to the output blob. This API facilitates the creation of a blob by appending blocks to it in parallel and if the sequence of the features is not critical this API can significantly improve the performance.

Additionally, each worker role puts a message on the termination indicator queue to indicate successful processing of the task. The web role keeps checking the number of messages in termination indicator queue to update the user interface with the current progress of the operation. Logically, when all of the tasks have been processed the number of messages in the termination indicator queue will match the number of base polygons. When this happens web role commits the resultant blob and flushes it as a persistent blob in the blob storage. The resultant blob becomes available for downloading or further processing, user interface is also updated with the URI of resultant blob. To commit a blob created using blocks the Azure API *PutBlockList* is used. In order to use *PutBlockList* it is necessary to provide the list of blocks to be committed, this list is maintained at the cloud end and can be downloaded by the web role by using another Azure API *GetBlockList*.

The queue storage mechanism provided by Azure platform comes handy for fault tolerance during processing. After a worker role reads a message from the task pool, the message disappears from the task pool for other worker roles and is subsequently deleted by the worker role after the processing ends successfully. In the event of a failure, the message does not get deleted and appears in the queue after a stipulated amount of time.

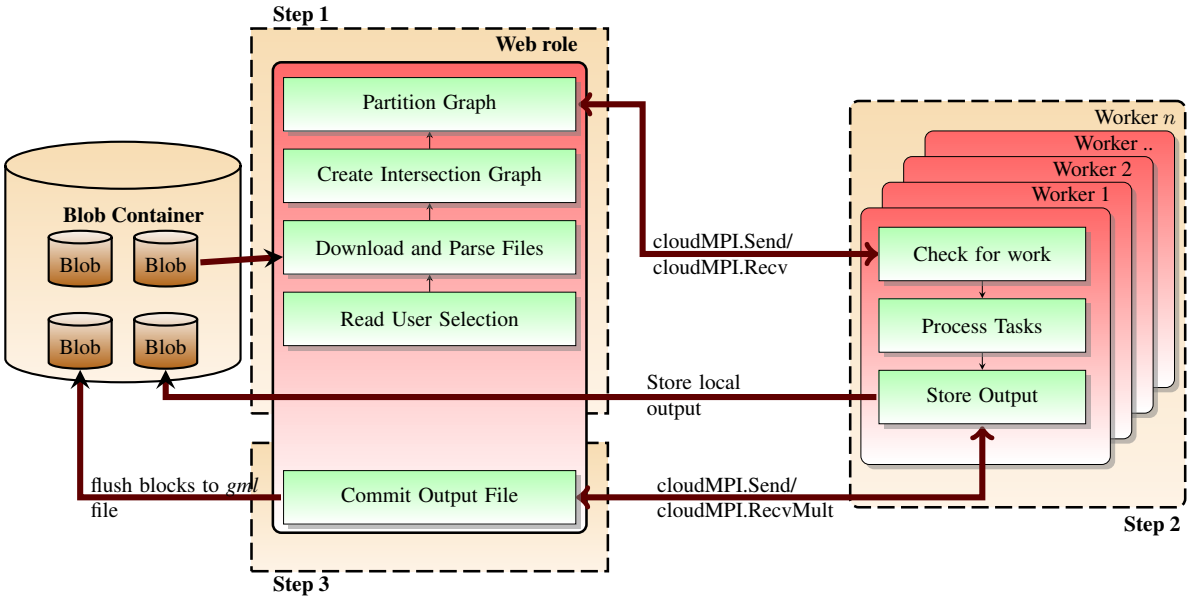


Figure 6.2 Software architecture of Crayons using cloudMPI

6.3.2 Crayons using cloudMPI

We have implemented the Crayons software using cloudMPI. It replaces all of the native Azure calls to cloudMPI calls. The architecture stays almost the same but direct interactions with the Azure storage are not needed any more. Figure 6.2 shows the architecture of the new Crayons software.

It is evident that the cloudMPI framework has resulted in a much simpler architecture of the Crayons software. Instead of concentrating on the complexities of the Azure cloud platform itself, application developers can now simply concentrate of their application logic. The results for Crayons are collected using the *RecvMult* API, which offers a much better performance than reading one message at a time.

An application could still need access to the Azure cloud storage, analogous to a traditional MPI application accessing the local file system for input and output modules. We have bundled a library that allows such access. Although this library requires the developer to have some familiarity with the Azure cloud platform's storage services, a simple file access

could be as simple as a REST call based on the resource URI. Therefore, we see this as a trivial challenge as most of the non-trivial communication happens after the data has been input and before results are written to the storage. This is the reason why in Figure 6.2 we have hidden the blob containers even though to download the files and to commit the output one call each should be made to the blob storage. We do not access any other storage service in the Cryaons software with the cloudMPI framework.

cloudMPI facilitates applications developers familiar with MPI a simple way to port their applications to the Windows Azure cloud platform. Crayons is an extremely complex piece of software, it took us over 1 year to get the entire workflow behaving as expected. However, porting Crayons using cloudMPI only took us little over 8 hours. The entire rewiring of the communication flow was a smooth process because of the intuitive APIs of the cloudMPI framework.

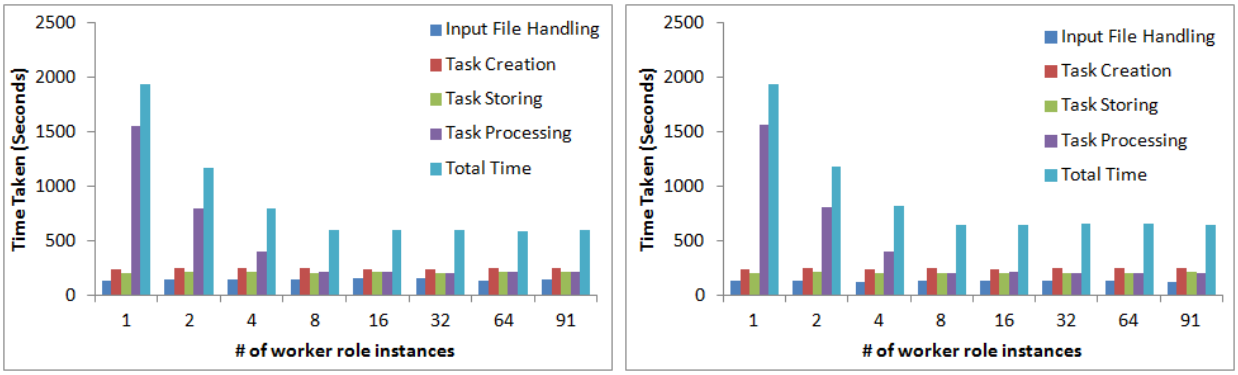
Replacing the native Azure API calls with simple cloudMPI routines, we are able to save over 700 lines of code. The cloudMPI library itself is about 2000 lines of code.

6.4 Performance and Results

In order to make sure that introducing an additional layer of code does not degrade the performance prohibitively, we assess the performance of Crayons software before and after cloudMPI. We would like to mention again that the goal of cloudMPI library is not to improve the performance. The motivation is to make it easier for eScience developers to write MPI-style applications for the Azure cloud platform.

Figure 6.3 shows the performance of Crayons software using cloudMPI and using native Azure APIs. It can be seen that the performance difference is trivial if any and such differences can be attributed to the dynamic nature of the Azure cloud platform. Factors such as number of tenants on a physical machine, network traffic, and distance between compute instances and data could introduce minor fluctuations.

The end-to-end time goes up by a little as the number of role instances increases. This could be attributed to the fact that in order to create a channel between master and slave



(a) Native Azure APIs

(b) cloudMPI APIs

Figure 6.3 Performance of Crayons with and without cloudMPI

processes, a separate queue needs to be created. As the number of processors goes up this time starts to increase. However, it is a one time process as once a queue has been created it is not automatically deleted on termination of the application.

6.5 Conclusion

In this chapter we have introduced cloudMPI framework for writing new MPI-style applications, as well as to port MPI based legacy applications to the Azure cloud platform. Our results and development times are a testimony to the fact that the cloudMPI framework improves a software developer's productivity and reduces the complexity of an HPC application. With only small amount of efforts required to port an application to the Azure cloud platform using cloudMPI, we believe that members of HPC community will find cloudMPI framework to be an essential tool for their research and development.

PART 7

CONCLUSIONS

Cloud computing is becoming mainstream for parallel and distributed computing. With the advent of public and private clouds, along with the ability to have on-premise cloud setup, cloud computing is poised to become the preferred platform for large-scale computing. The current roadblocks for adoption of cloud computing are the non-intuitive nature of the architecture as well as the lack of uniform standards among various vendors. There are no familiar tools that are supported across the spectrum.

In this thesis we have introduced newer frameworks for accelerated application writing for cloud platforms. Our results and development times are a testimony to the fact that the frameworks are highly effective at increasing the productivity of the application developers and thus can greatly help with adoption of cloud computing at large scale.

Our cloudMPI framework makes it easier for researchers to not only write applications, but to port legacy MPI applications to the Azure cloud platform without sacrificing the performance. With the meager amount of efforts required to port applications to the Azure cloud platform using our frameworks, we believe that the researchers will be motivated to explore cloud platforms for their research.

We have also implemented a complex application from the GIS domain to parallelize the end-to-end map overlay process over polygonal data. Although our speedups are decent, we believe that splitting large polygons to effectively partition that intersection graph will minimize the partitioning and communication overheads and maximize the load balance. Currently, output of the *Crayons* system is a GML file that can be downloaded for further processing. As a future work, creating an interactive graphical interface to render maps that allows users to visually interact with the output file will be a good addition to the *Crayons* system. Additionally, *Crayons* currently only supports single user requests for

spatial operations. Extending *Crayons* such that users can register, get dedicated resources for a chosen amount of time, assign tasks, and get notifications on completion will improve its usability.

REFERENCES

[1] Windows azure platform at a glance. [Online]. Available: http://blogs.msdn.com/b/jmeier/archive/2010/02/16/windows-azure-platform-at-a-glance.aspx

[2] FPA, "Gis thematic layers and datasets," http://www.fpa.nifc.gov/Library/Documentation/FPA_PM. December 2011.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[4] C. A. Lee, "A perspective on scientific cloud computing," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 451–459.

[5] S. L. Garfinkel, "An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS," Tech. Rep.

[6] P. Mell and T. Grance, "The NIST Definition of Cloud Computing, Special Publication 800-145," 2010.

[7] L. Wang, J. Tao, M. Kunze, A. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, sept. 2008, pp. 825 –830.

[8] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," *ArXiv e-prints*, vol. 901, Dec. 2009.

[9] B. Rimal, E. Choi, and I. Lumb, "A Taxonomy and Survey of Cloud Computing Systems," in *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, aug. 2009, pp. 44 –51.

[10] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, sept. 2008, pp. 5 –13.

[11] A. Thakar and A. Szalay, "Migrating a (large) science database to the cloud," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 430–434.

[12] G. Turcu, I. Foster, and S. Nestorov, "Reshaping text data for efficient processing on amazon ec2," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 435–444. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851540

[13] SYS-CON, "Twenty experts define cloud computing," http://cloudcomputing.sys-con.com/read/612375_p.htm, 2008.

[14] Google Trends, "Google trends," http://www.google.com/insights/, November 2011.

[15] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, pp. 599–616, June 2009.

[16] J. J. Rehr, F. D. Vila, J. P. Gardner, L. Svec, and M. Prange, "Scientific computing in the cloud," *Computing in Science and Engineering*, vol. 12, pp. 34–43, 2010.

[17] S. Srirama, O. Batrashev, and E. Vainikko, "Scicloud: Scientific computing on the cloud," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Clus-*

ter, Cloud and Grid Computing, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 579–580.

[18] P. Ross, "Cloud Computing's Killer App: Gaming," *Spectrum, IEEE*, vol. 46, no. 3, p. 14, march 2009.

[19] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the use of cloud computing for scientific workflows," *2008 IEEE Fourth International Conference on eScience*, pp. 640–645, 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4736878

[20] [Online]. Available: http://whirr.apache.org

[21] [Online]. Available: http://www.jclouds.org

[22] R. A. A. Bruce, S. Chapple, N. B. MacDonald, A. S. Trew, and S. Trewin, "Chimp and pul: Support for portable parallel computing," *Future Generation Computer Systems*, vol. 11, no. 2, pp. 211–219, 1995.

[23] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, 1990.

[24] C. The MPI Forum, "Mpi: a message passing interface," in *Proceedings of the Conference on High Performance Networking and Computing*, 1993, pp. 878–883.

[25] [Online]. Available: http://awards.computer.org/ana/award/viewPastRecipients.action?id=16

[26] [Online]. Available: http://mvapich.cse.ohio-state.edu/

[27] [Online]. Available: http://software.intel.com/en-us/articles/intel-mpi-library/

[28] [Online]. Available: http://msdn.microsoft.com/en-us/library/bb524831%28v=vs.85%29.aspx

[29] [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[30] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[31] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster: evaluating amazon cluster compute instances for running mpi applications," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 11:1–11:10.

[32] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey, "Early observations on the performance of windows azure," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 367–376. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851532

[33] D. Agarwal, S. Puri, X. He, and S. K. Prasad. Crayons - a cloud based parallel framework for GIS overlay operations. [Online]. Available: http://cs.gsu.edu/dimos/crayons.html

[34] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[35] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 143–157. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043571

[36] FGDC, "National Spatial Data Infrastructure: Federal Geographic Data Committee," http://www.fgdc.gov/nsdi/nsdi.html, 1994.

[37] OGC, "OpenGIS Standards and Related OGC documents: OGC," http://www.opengeospatial.org/standards, 1994.

[38] WCS, "Web Coverage Service - OGC," http://www.opengeospatial.org/standards/wcs, 2007.

[39] GeoRSS, "Main Page: GeoRSS," http://www.georss.org/Main_Page, 2009.

[40] OJWS, "OnEarth, JPL WMS Server," http://onearth.jpl.nasa.gov/, 1936.

[41] Census.gov, "Us census data," http://www.census.gov/, December 2011.

[42] GDOT, "Georgia department of transportation," http://www.dot.state.ga.us/Pages/default.aspx, 1916. [Online]. Available: http://www.dot.state.ga.us/Pages/default.aspx

[43] USGS, "U.s. geological survey," http://www.usgs.gov/, 1879.

[44] NASA, "Jet propulsion laboratory," http://www.jpl.nasa.gov/, 1936. [Online]. Available: http://www.jpl.nasa.gov/

[45] Open Topography Facility, "Open topography," http://opentopo.sdsc.edu/gridsphere/gridsphere?cid=

[46] W3C, "W3C Geospatial Incubator Group," http://www.w3.org/2005/Incubator/geo/, July 2006.

[47] S. Dowers, B. M. Gittings, and M. J. Mineter, "Towards a framework for high-performance geocomputation: handling vector-topology within a distributed service environment," *Computers, Environment and Urban Systems*, vol. 24, no. 5, pp. 471 – 486, 2000.

[48] P. Richmond, S. Coakley, and D. M. Romano, "A high performance agent based modelling framework on graphics card hardware with CUDA," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, ser. AAMAS '09. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1125–1126.

[49] HAZUS-MH, "Hazus-MH Overview," May 2011.

[50] J. Li, M. Humphrey, D. Agarwal, K. Jackson, C. van Ingen, and Y. Ryu, "eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –10.

[51] J. R. Delaney and R. S. Barga, *The Fourth Paradigm: Data Intensive Scientific Discovery*. Microsoft Research,, 2009, ch. Observing the Oceans - A 2020 Vision for Ocean Science.

[52] J. Bentley and T. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *Computers, IEEE Transactions on*, vol. C-28, no. 9, pp. 643 –647, sept. 1979.

[53] B. Chazelle and H. Edelsbrunner, "An optimal algorithm for intersecting line segments in the plane," *J. ACM*, vol. 39, pp. 1–54, January 1992.

[54] T. M. Chan, "A simple trapezoid sweep algorithm for reporting red/blue segment intersections," in *In Proc. 6th Canad. Conf. Comput. Geom*, 1994, pp. 263–268.

[55] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005.

[56] P. K. Agarwal, L. Arge, T. Mølhave, and B. Sadri, "I/O-efficient efficient algorithms for computing contours on a terrain," in *Proceedings of the twenty-fourth annual symposium*

on Computational geometry, ser. SCG '08. New York, NY, USA: ACM, 2008, pp. 129–138.

[57] X. Zhou, D. Truffet, and J. Han, "Efficient polygon amalgamation methods for spatial olap and spatial data mining," in *Advances in Spatial Databases*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1999, vol. 1651, pp. 167–187.

[58] J. D. Hobby, "Practical segment intersection with finite precision output," *Computational Geometry*, vol. 13, no. 4, pp. 199 – 214, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925772199000218

[59] R. G. Healey, M. J. Minetar, and S. Dowers, Eds., *Parallel Processing Algorithms for GIS*. Bristol, PA, USA: Taylor & Francis, Inc., 1997.

[60] F. Wang, "A parallel intersection algorithm for vector polygon overlay," *Computer Graphics and Applications, IEEE*, vol. 13, no. 2, pp. 74 –81, mar 1993.

[61] [Online]. Available: http://www.cs.man.ac.uk/~toby/alan/software/gpc.html

[62] B. R. Vatti, "A generic solution to polygon clipping," *Commun. ACM*, vol. 35, pp. 56–63, July 1992.

[63] P. van Oosterom, "An R-Tree based Map-Overlay Algorithm," in *EGIS/MARI'94*, 1994.

[64] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "A system for GIS polygonal overlay computation on linux cluster - an experience and performance report," in *IEEE International Parallel and Distributed Processing Symposium workshops, to appear*, Shanghai, China, May 2012.

[65] A. Guttman, "R-Trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, pp. 47–57, June 1984.

[66] H. Li, *Introducing Windows Azure*. Berkely, CA, USA: Apress, 2009.

[67] D. Agarwal and S. K. Prasad, "Azurebench: Benchmarking the storage services of the Azure Cloud Platform," in *IEEE International Parallel and Distributed Processing Symposium workshops*, Shanghai, China, May 2012.

[68] D. Agarwal, S. Puri, X. He, and S. K. Prasad, "Cloud computing for fundamental spatial operations on polygonal gis data," in *Cloud Futures 2012 - Hot Topics in Research and Education*, Berkeley, California, United States, May 2012.

[69] D. Agarwal and S. K. Prasad, "Lessons learnt from the development of gis application on azure cloud platform," in *IEEE CLOUD*, 2012, pp. 352–359.

[70] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 261–276.

[71] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.

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

[73] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273005

[74] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818.

[75] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, and J. Sifakis, "A framework for automated distributed implementation of component-based models," *Distributed Computing*, pp. 1–27.

[76] J. Hoffert, D. Schmidt, and A. Gokhale, "Adapting distributed real-time and embedded pub/sub middleware for cloud computing environments," in *Middleware 2010*, ser. Lecture Notes in Computer Science, I. Gupta and C. Mascolo, Eds. Springer Berlin / Heidelberg, 2010, vol. 6452, pp. 21–41.

[77] J. Ekanayake, J. Jackson, W. Lu, R. Barga, and A. Balkir, "A scalable communication runtime for clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, july 2011, pp. 211 –218.

[78] T. Gunarathne, T.-L. Wu, J. Qiu, and G. Fox, "Cloud computing paradigms for pleasingly parallel biomedical applications," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 460–469.

[79] E. Mocanu, V. Galtier, and N. Tapus, "Generic and fault-tolerant bag-of-tasks framework based on javaspace technology," in *Systems Conference (SysCon), 2012 IEEE International*, march 2012, pp. 1 –6.

[80] A. Fabbro, "Linux on azure-a strange place to find a penguin," *Linux J.*, vol. 2013, no. 226, Feb. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2457428.2457430

[81] [Online]. Available: https://cloud.google.com/products/compute-engine

[82] D. Petcu, "Portability and interoperability between clouds: Challenges and case study," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, W. Abramowicz, I. Llorente, M. Surridge, A. Zisman, and J. Vayssire, Eds. Springer Berlin Heidelberg, 2011, vol. 6994, pp. 62–74. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24755-2_6

[83] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running hpc applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 395–401. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851535

[84] D. Agarwal and S. K. Prasad, "Azurebot: A framework for bag-of-tasks applications on the azure cloud platform," in *to appear, IEEE International Parallel and Distributed Processing Symposium workshops*, Boston, USA, May.