Building Cloud Applications

Contents

1 Introduction **2**

2 Considerations on Azure that impact the implementation **2**
2.1 Lack of MPI . . . 2
2.2 Azure Storage and the shared memory abstraction . . . 4
2.3 Queues, jobs, tasks and QueueServices . . . 4
2.4 Affinity between workers and storage . . . 5
2.5 Idempotency . . . 5
2.6 Workers are at first tasks agnostic and stateless . . . 6
2.7 Scaling up is a developer initiative . . . 6
2.8 There are no strict guarantees on the number of workers actually running . . . 7
2.9 Pinging queues as a tradeoff between cost and simplicity . . . 7
2.10 Monitoring how many messages are delivered . . . 7
2.11 Workers Communication . . . 8
2.12 Atomicity in the BlobStorage . . . 9
2.13 Lokad-Cloud . . . 9

3 The counter primitive **9**
3.1 Sharded Counters . . . 9

4 The example of a naïve MapReduce **9**

1 Introduction

The SaaS cloud solutions have already proved to be a successful economical model, as demonstrated for example by Amazon or Gmail. In parallel, the IaaS cloud solutions have also proved to be good candidates for many customers, as suggested by the long list of some of Amazon Web Service customers[1] or by their business volume (see [?]).

The PaaS cloud solutions are reputed to be easy to manipulate. Indeed, many tools are provided to improve the developer efficiency. In addition to these tools, many abstractions and primitives are already available to prevent the customer from re-implementing the same engineering solutions again and again. For example, the Azure Queues provide a synchronization mechanism that helps building loosely coupled components at a low development cost. To our knowledge the Azure platform has not yet been originally designed in the specific goal to host intensive computations. While the overall system has been designed to provide very satisfactory scale-up (whether in term of the number of computing units available or aggregated bandwidth) in order to guarantee scale-up for many cloud applications, the Azure PaaS system has been originally targeted to business cloud applications, not to scientific applications. In this chapter, we investigate to what extent is the Paas platform Azure well-suited to scientific computations.

Since Azure has not been initially designed to be a scientific computing platform, no computation framework (with the exception of Windows Azure HPC introduced in the introduction of chapter **??**) is presently available for Azure: Azure provides neither a "low-level" framework such as MPI, nor a "high-level" framework such as MapReduce. While the Azure team has reported that they are working on implementing a MapReduce framework for Azure, this framework is not already available.

In this situation, the storage abstractions and the update primitives (see for example section **??**) that are suited for most cloud applications can turn out to be inefficient for computation-intensive applications. During my thesis work, I've been involved in four of these computation-intensive applications. Two of them are the cloud Batch K-Means prototype and the cloud Vector Quantization prototype[2] that are described in detail in chapters **??** and **??**.

[1] http://aws.amazon.com/solutions/case-studies/

[2] Both of these prototypes are available at http://code.google.com/p/clouddalvq/

2 Considerations on Azure that impact the implementation

2.1 Lack of MPI

The section **??** has highlighted the importance of MPI for the distributed Batch K-Means on DMM architectures. More specifically, the various MPI implementations provide an abstraction layer that disburden the application developer from manual management of inter-machines communications. The section **??** has showed how the MPI primitives actual implementation has a significant impact on the speed-up that can be achieved by the distributed Batch K-Means on DMM architectures: this is the tree-like topology of the communication patterns of MPI primitives that results in a $O(logM)$ cost for averaging the prototypes versions.

As explained in section Azure Compute (section **??**) of chapter **??**, an internal endpoint is exposed by each processing unit of Azure (the role instances) so that each of the processing unit could talk directly with the other processing units using a low-latency, high-bandwidth TCP/IP port. The bandwidth of these direct communications is very sensitive to multiple factors like the number of units that are communicating at the same time (because of aggregated bandwidth boundaries), the fact the resource manager may allocate other VM instances from other deployments (applications) in the same physical hardware, the fact these other VM may also be I/O intensive, etc. The average behavior of such direct communications is yet very good: [9] and [1] reports direct inter-machines communication bandwidth from 10 MB/sec to 120 MB/sec with median measurement bandwidth of 90MB/sec.

A key feature of Azure VM system is that the cloud client cannot get any direct topology information about the VM it temporary owns. On the contrary, the VM are totally abstracted to disburden the cloud client from these considerations. Such a design does not prevent to build a MPI-like API to automatically harness direct inter-machines communications but it makes more difficult to build such an efficient framework. Several works have ported MPI on cloud computing platform such as EC2 (we refer the reader for example to [3] or [2]).

Because of the importance of bandwidth in the speed-up performance of distributed Batch K-Means (see section **??**), and because the direct inter-machines bandwidth is much higher than the bandwidth between the storage and the processing units (see section **??** of chapter **??**), MPI would be a very efficient way to run Batch K-Means on Azure. To our knowledge, such a framework has not yet

been provided for this platform.

UN PARAGRAPHE PR JUSTIFIER POURQUOI JE L'AI PAS IMPLÉMENTÉ ?

2.2 Azure Storage and the shared memory abstraction

The Azure Storage service provides a shared memory abstraction in the form of the BlobStorage (and the TableStorage, but because of their similarity we will only mention BlobStorage in this section). The different processors of a SMP system can access the shared memory RAM to read or write data and therefore use this shared memory as a communication media between the different processors. In the same way, all the different computing units of an Azure application can access any of the blobs stored into the BlobStorage.

There is however a critical difference between the use of shared memory in a SMP system and the use of BlobStorage as a shared memory in a cloud OS environment: it is the bandwidth. Indeed, in the SMP case, each processor is geographically very close to the shared memory and efficient communication media such as buses are used to convey the data between the shared memory and the processors. On the contrary, the BlobStorage data are stored on different physical machines than the ones hosting the processing VM. Therefore, the communication between the computing units and the BlobStorage are conveyed through TCP/IP connections between the distant machines. The resulting bandwidth are much lower than the bandwidth of a real shared memory: around 10MB/sec for reads and 3 MB/sec for writes (see section **??** of chapter **??**)).

2.3 Queues, jobs, tasks and QueueServices

Following the terminology of the original MapReduce research paper ([4]), the total execution of an algorithm is referred as a job. A given job is divided in multiple tasks that stands for the elementary blocks of logic that are executed. Each task is run by a single worker, and a single worker can only process one task at once. During the duration of an entire job, each processing unit is expected to run successively one or multiple tasks.

In the event of the number of tasks of a given job being lower than the number of processing units, or in the event of one or several workers being temporary isolated, some workers may process no tasks during a given job.

The multiple tasks are described by items stored in queues. More specifically, each tasks that need to be run is put in a queue of the QueueStorage. When a processing unit is available, it pings a queue and un-queue an item that stands for a specific task. When the task is completed, the corresponding processing unit pings a queue to get a new item and then process the corresponding task. The number of tasks does not need to be defined when the job is started. On the contrary, a common Azure application design is that many of the tasks that are completed produce one or several new tasks.

Among all the tasks that are achieved during a given job, many of them refer to the same logical operations applied on multiple distinct data chunks of the same type, a situation often described in the literature as data level parallelism. To reflect this data parallelism, a frequently chosen design consists in gathering in the same queue only the items referring to the same specific logical operation but applied on distinct data chunks. In such a design, to each queue is associated a QueueService which holds the logical operations to be applied on each item of the corresponding queue.

2.4 Affinity between workers and storage

Azure is not providing any affinity between data and workers. Data are abstracted into BlobStorage and TableStorage and it is impossible by design to try to run jobs on CPUS that are physically near the place data are stored. CITE GOOGLE MAP REDUCE. LOOK FOR XMPP PROTOCOL.
Presently, there are no mechanisms in Azure that provides the same affinity between the workers and the storage than MapReduce does. This means that we cannot push some jobs where the data is stored since we can't know on which machine a data piece is stored. Each worker processing data will need to first download the data it needs to process, on the contrary of Google's Map Reduce, where the jobs are scheduled and assigned on each worker by the framework in such a way that workers downloading data will be minimized.
This is intended by Microsoft (CITATION NEEDED) to simplify development of cloud apps : one does not need to design its application to take data location into account, Azure providing a satisfactory scale-up and bandwidth. This choice is about simplicity over performance.

2.5 Idempotency

Let's consider the general problem of message processing. Some worker will consume a message and then update some data stored in a durable way (for example in

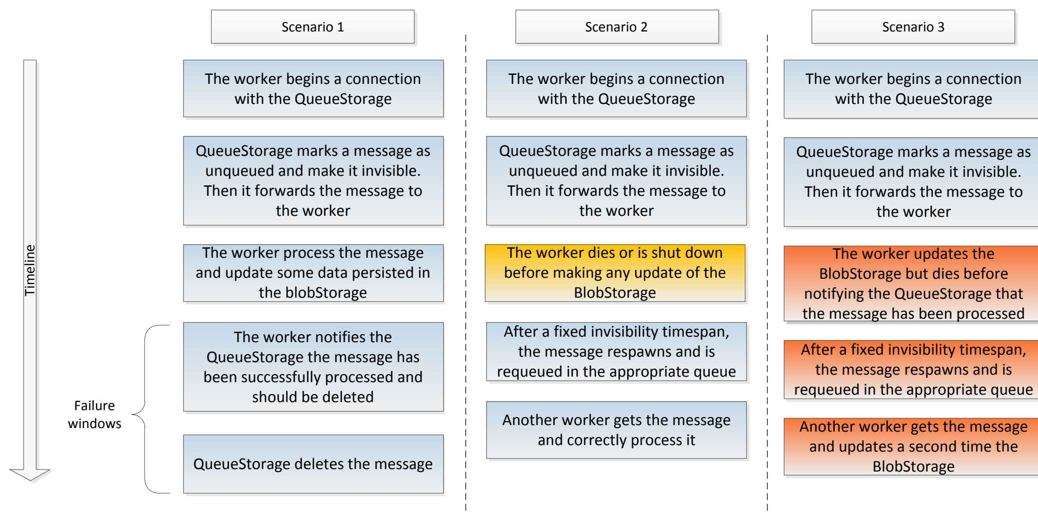

Figure 1: Some scenario of data updates.

BlobStorage). The message is consumed by the worker, communication is made between the worker and storage, data is updated in the storage, but the worker dies before it acknowledges the queue that the message has been processed and should be deleted. Queues have been designed to react to machine failures as a "at-least-once" messaging device : in a failure, message is requeued after some timespan and another worker will process the same message, resulting in a second update operation being performed on the same data chunk.

Technically, this issue derives from the fact there are no direct mechanism to couple the message consumption and deletion with the update of the persistent data chunk in such a way that one of these two events cannot happen without the other. The absence of this coupling leads to failure windows between the end of data update and message deletion in which the message is delivered and processed more than once.

2.6 Workers are at first tasks agnostic and stateless

same CPU, no affinity with the storage at first, non-stateless and idempotency

2.7 Scaling up is a developer initiative

Through the management API, administrator can change the number of workers available. Then, we are no more in a fixed hardware framework as K-Means run in other papers experiments. We can modulate the hardware so it better fits our need. Therefore we can ask a new question : how many workers do we need to minimize the whole time of the algorithm ?

2.8 There are no strict guarantees on the number of workers actually running

Due to the huge amount of time necessary to set up workers [9] (which is going to be improved), workers will be instanciated before the experiments are run and their number is supposed to be constant over time. Yet, since a worker can be shutdown, a VM deplaced or paused arbitrarily by Azure without notification, the number of workers is not guaranteed to be exactly the number requested at each instant.

2.9 Pinging queues as a tradeoff between cost and simplicity

Workers will be pinging queues to detect if there are some messages to process. Since even QueueStorage operations are charged, we can wonder whether pinging queues can be avoided, and whether it is affordable. Indeed, we could open a TCP connection on each worker, set-up a listener so each worker is listening for notification. Each time a notification is sent, the listener tells the worker to ping the queue and get the message, or the notification is the message by itself and worker is directly told to process the message. In the first case, we do not ping queues unless a message needs to be processed. In the second case, we do not ping queues at all. We have been choosing to not implement listeners patterns and TCP connections, for simplicity reasons. Pinging a queue every 100 ms for a worker means a 315 millions request per worker per year, charged as 3000 dollars (see [**?**]). We have been adopting a back-off strategy : each time we ping an empty queue, we double time before pinging again the queue CHECK THIS IN LOKAD.CLOUD. IN THE EXPERIMENTATION PART, GIVE HOW MANY TIMES THE QUEUE IS PINGED

2.10 Monitoring how many messages are delivered

It is not possible to get an exact count of how many messages are stored in some queue, we only can get an approximative count of queued messages when asking to the queue its metadata. When a service needs to process P messages, then

need to push a message in another queue to launch a new service, we need to get an exact count of remaining messages to process. To achieve this, we are using blobcounters, blobs storing an integer and decremented by each worker using etag properties when the worker is done with a message. Since the worker can fail at any moment, and a failure means the message is requeued after some delay and is going to be processed again, we decrement the counter in the end, when the job is finished and the message is going to be deleted. Thus, the only way a single message can lead to decrement twice the counter is to make the worker fail exactly when it is noticing the queue to delete the message. Theoretically, the counter could therefore be decremented twice (with a very low probability). One way to deal with this issue would be to use an idempotent counter :

1. we push P messages in the queue. Each message has a unique Id between 1 and P

2. we push a binary counter of P bits set to 1111..1111 in the blobStorage

3. each time the job stored in message i is completed, we update using etag the counter and apply to the counter a & operation 111110111111 where 0 is at index i. This way we made a counter where decrements will be idempotent.

Even if getting number of messages stored in a queue is approximate, Azure is providing a transactional count of how many times a message is dequeued. FINISH HERE

2.11 Workers Communication

Azure does not offer currently any standard API for distributed computation, neither a low level one such as MPI [8], nor a more high level one such as Map Reduce [4] or Dryad [5]. Map reduce could be implemented using Azure components (following the strategy of [6]), yet, as pointed out in e.g. [7], those high level API might be inappropriate for iterative machine learning algorithms such as the k-means or learning vector quantization because of MapReduce overhead on synchronisation and waiting process.
We therefore need to explicitly design our communication process between Azure workers, process that can realized by 2 means :

1. Workers can communicate each other through direct IP communications, that is to say re-implement a peer-to-peer network inside Azure cloud. TO DEVELOP

2. Azure storage system can be used to implement synchronization and communication between workers : each worker is pinging (or "listening") a

specific queue, and it gets some message from this queue holding some blobname in BlobStorage where it can retrieved the data it needs.

While the first solution comes with higher performance in terms of bandwidth, as reported in AzureScope or ... (CITATION NEEDED), it implies a very tough work which would be like implementing MPI on Azure, which was outside the scope of this thesis. Moreover, the "storage approach" for communication is much more in agreement with Azure's abstractions and design thinking (CITATION NEEDED). Thus, we will rely therefore directly on Azure queues and blob storage to build all our communication systems.

2.12 Atomicity in the BlobStorage

2.13 Lokad-Cloud

Lokad-Cloud[3], an open-source framework that adds a small abstraction layer to ease Azure workers startup and life cycle management, and storage access.

3 The counter primitive

3.1 Sharded Counters

4 The example of a naïve MapReduce

References

[1] Azure scope. `http://azurescope.cloudapp.net/`.

[2] Mpi cluster on ec2. `http://datawrangling.s3.amazonaws.com/elasticwulf_pycon_talk.pdf` read the 03/05/2012.

[3] Dan C.Marinescu. *Cloud Computing: Theory and Practice*.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[3] `http://code.google.com/p/lokad-cloud/`

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

[6] Huan Liu and Dan Orban. Cloud mapreduce: a mapreduce implementation on top of a cloud operating system. Technical report, Accenture Technology Labs, 2009. `http://code.google.com/p/cloudmapreduce/`.

[7] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[8] Marc Snir, Steve Otto, Steven Huss-Lederman, Walker David, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Boston, 1996.

[9] Ming Mao Arkaitz Ruiz-Alvarez Zach Hill, Jie Li and Marty Humphrey. Early observations on the performance of windows azure.