Performance evaluation of AWS

Exploring storage alternatives in Amazon Web Services

Jesús Hernández Martin, Ioan Raicu
Data-Intensive Distributed Systems Laboratory
Illinois Institute of Technology
Chicago, USA
jherna22@hawk.iit.edu, iraicu@cs.iit.edu

***Abstract*— with the increasing number of public cloud platforms and the growth in terms of computing capacity, I/O performance must be one of the main points to look at so that it keeps up with the current compute capacity. In order to start considering any of the existing public cloud platforms for its use in scientific or in general, any high I/O demanding application, we have to study their raw performance in terms of I/O. In this paper, I focus on the most known IaaS cloud platform nowadays: Amazon's public cloud. Its ease to use, reliability and various API interfaces allow anyone willing to outsource their compute/storage capacity to the cloud without any issues. This document describes the tools I used for benchmarking, a description of the file systems and storage solutions involved in this study and finally, the results I obtained from running all the benchmarks.**

***Index Terms*—storage, Amazon, S3, EBS, pvfs, nfs, dynamoDB**

I. Introduction

Nowadays, all the most demanding scientific applications and simulations are run on top of big dedicated parallel systems, which are usually not accessible to everyone. These computers are getting bigger and faster by the year, so as to be ready to run new compute intensive scientific applications. These applications not only require large amounts of computing power but also need to have access to large datasets.

Nevertheless, over the last few years we have seen how new trends in the field of distributed systems are gaining importance with the concept of Cloud Computing. In this regard, Amazon.com introduced AWS (Amazon Web Services) in 2006, which nowadays is the largest IaaS public cloud platform in the market.

AWS offers anyone the opportunity to use their computing infrastructure in a pay-per-hour basis. Unlike big dedicated datacenters, AWS's infrastructure works on top of commodity hardware, offering both dedicated and virtualized resources, depending on the users' needs. Initially, AWS was meant to be used by web applications to provide their services at a given cost. However, the need for building highly parallel systems puts AWS in the spotlight of several members within the scientific community, wondering if these cloud platforms could prove to be a good alternative to substitute current HPC systems. So far, studies are being made in order to corroborate whether a virtualized cluster can keep up to the task when running high capacity demanding applications.

If we talk about capacity, we cannot ignore I/O. I/O on parallel computers has always been slow compared with computation and communication. As computers get larger and faster, I/O becomes even more of a problem, to the point that when the technology reaches exascale, the bottlenecks of I/O will be dramatic with the existing level of development [1]

Applications running on most multicore platforms are usually held back by storage systems that cannot keep up. Although HDDs provide the capacity needed to handle large amounts of data, their I/O performance capabilities are relatively slow. In fact, storage system I/O performance has increased by only a small fraction of server performance, which seems to be driven largely by Moore's Law.

The main concern of this document is to explore the different storage options offered by Amazon Web Services, providing insight into their raw I/O performance and suitability for their usage in scientific applications.

II. Amazon EC2

Amazon Elastic Compute Cloud (EC2) [2] is a web service that allows anyone to run their own applications on Amazon's computing infrastructure, by letting customers "rent" computing resources by the hour.

Clients are given access to an "unlimited" source of compute capacity, which is delivered through what is known as EC2 instance. Basically, an instance is a running machine on Amazon's cloud platform. Each of these instances is deployed with an Amazon Machine Image (AMI), which is just a pre-configured operating system and some bundled application software. There exist several types of instances, each of them with different compute capacities, memory size, I/O performance and storage.

If we consider the way we can have access to these instances, we can categorize them in three different types:

- Reserved instances: Amazon allows us to pay upfront per each instance that we want to use during a given period of time, and in exchange, they give us a lower hourly cost for each of them. Along with the savings, with these instances we make sure that we will have availability through all the period that we paid for.
- On demand instances: these are the most common type of instances. You only pay for what you use, allowing easy allocation and deallocation of

resources, depending on your capacity requirements. Customers are billed at the end of each month.

- Spot instances: this is a very interesting concept. In order to achieve a better utilization of their infrastructure, Amazon allows us to bid on unused EC2 capacity and run instances until the current spot instance price exceeds our bid. The spot price is set by Amazon based on the available capacity and load of their systems and it is updated in a 5 minute period. The prices of these instances are much lower than what you pay for On-demand instances. As a drawback, the availability of you instances is only assured while the spot price is under bid. As previously stated, Amazon automatically terminates those instances whose bid is exceeded by the spot price. Besides, one cannot stop a spot instance and use it later as it happens with on-demand or reserved instances. Spot instances can only be terminated or rebooted.

Among these types, the spot instances seem to be the most appropriate for running short-term applications under certain conditions, since they provide the same capacity and features as the other instances at a lower rate. These include scientific applications, which usually run for a predictable amount of time, lowering the costs per experiment.

III. STORAGE ALTERNATIVES IN AWS

There are several types of storage options in AWS, each of them with different features which make them more suitable for one or another application. When you rent an instance in Amazon EC2, you basically have three ways to store your files:

- Elastic Block Store (EBS) volumes. These are network attached volumes that can be mounted to a device in an EC2 instance and interact with them as if they were mounted locally. These volumes are dynamically created, so one can choose its size (from 1GB to 1TB) and decide whether you want it pre-loaded with an existing image (for example a dataset).
  EBS volumes are billed $0.10/GB per month and $0.10 per 1 million I/O requests to them.
  According to Amazon, all the data stored in EBS volumes is implicitly replicated across multiple servers within the same availability zone, which makes them highly reliable in comparison with standard hard drives.
  Since these volumes are network attached, they have a theoretical throughput limit, which is given by the instance's network bandwidth (1Gbps in most cases)

- Simple Storage Service (S3): built from commodity hardware, S3 is the storage choice for those who require speed, scalability and security at the same time. Unlike EBS, this cannot be mounted to an EC2 instance without the help of some middleware (like s3fs). However, it provides higher availability and redundancy, since data stored in S3 is replicated across different servers in different availability zones.
  S3 is very suitable for applications which require high scalability and bandwidth. S3 bandwidth is constrained by the user accessing S3, not by S3 itself, thus providing "infinite" bandwidth to its users.
  The high-level definition for a file in S3 is an "object" and each object is stored in a bucket, which can be chosen among different availability zones.
  Its price depends on the data to be stored, the output bandwidth, number of requests (S3 is accessed through its SOAP/REST API) and redundancy (Amazon offers reduced redundancy S3 storage, which is cheaper).

- Instance store
  By default, all instances except for the t1.micro are provided with some amount of instance storage. This storage is physically attached to the host computer, which may be shared by several VMs at the same time. Hence, the instance store subsystem may also be shared by the different VMs running on the same machine, although each VM has exclusive and dedicated access to its own instance store.
  Unlike the previous storage options, instance storage is not persistent and the data contained in it may be lost if the VM to which it is attached is stopped or terminated.
  The size of this instance store varies from instance to instance, ranging between 150GiB to 3.3 TiB. The same applies to its bandwidth, which varies depending on the type of instance.
  The price of this storage is included within the cost of the instance rental, so there are no extra charges per GB or bandwidth.

IV. NFS

Despite its age, I considered that NFS was a good start point for the distributed file systems benchmarks. NFS (network file system) allows remote hosts to mount file systems over a network and interact with those file systems as though they are mounted locally. The main issue of NFS is that the performance of the file system is constrained by the network capacity of the central server, in which all the files are stored.

V. PVFS2

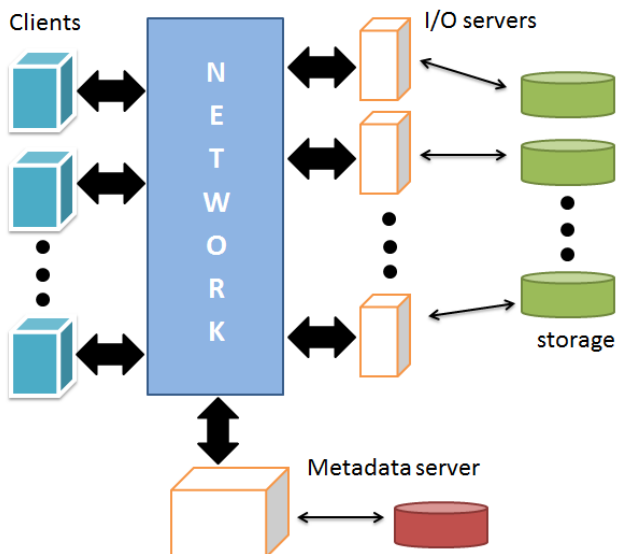
PVFS stands for Parallel Virtual File System (PVFS) [3]. It is an open source parallel file system, aimed at providing a scalable and high-performance parallel file system on top of a Linux based cluster. PVFS is designed so that the applications that access this file system have their data spread out across

different nodes (local disks) within the cluster in which PVFS is installed. To achieve this, PVFS relies (partially) on the network interface of each node, routing incoming byte streams to the different nodes.

By using PVFS, an application running on node X which requires some kind of I/O operation will not have to wait until a previous application finishes performing I/O operations on the same local storage drive, since its byte stream will be re-routed by PVFS through the network to another node Y in the same cluster which may be idle.

We can find three different elements in PVFS:

- I/O servers: store the data in their local storage drives
- Metadata server: stores the information of all the files spread across the parallel file system
- Clients: store and retrieve data from the servers.



Among the interfaces provided in PVFS, we can find ROMIO, an MPI-I/O interface implementation that is detailed in the next section.

VI. MPI-IO ROMIO IMPLEMENTATION

MPI-I/O is the parallel I/O interface included in the MPI-2 specification [4]. It was developed to overcome the lack of portability and optimization that POSIX had for parallel I/O.

Based on the coordination between processes, there are different data access patterns in MPI-IO. Independent routines are used when there is only one processor and an I/O request, or different processors accessing different files. Collective routines involve more than one processor. In a collective call, all the processors open the same given file, but each of them has a different view of the file. This view defines the data that is visible to each processor. Hence, collective routines usually perform better than independent routines, since a number of small requests to the same file can be merged into one big request in order to improve I/O performance.

Besides, MPI provides three different types of positioning within a file: individual file pointers, in which each processor increments its own pointer after a write/read operation; shared file pointers, in which a unique file pointer is shared between all the processes and explicit offsets, by means of which each process writes/read at the position specified by the offset.

VII. TOOLS

In order to make the benchmarks as exhaustive as possible, it is important to use wide-spread benchmarking tools. For my study, I considered Bonnie64 [5], hdparm [6] and IOR [7]. After several test, I decided to go with IOR. However, in some cases I have had to develop my own benchmarking tools, since I could not find any well-known tool which fulfilled my needs and that was also widely accepted.

A. IOR

To provide comprehensive results, I decided to use IOR. IOR is a benchmark tool used for testing the performance of parallel file systems using various interfaces and access patterns. At the same time, IOR is very flexible in terms of customization, accepting a high variety of input parameters.

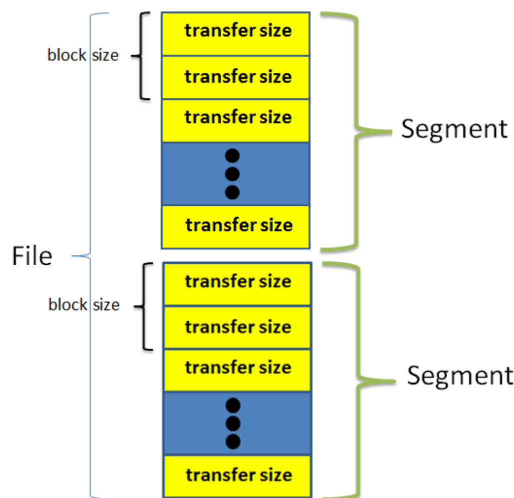
For my study, I have used both the POSIX and MPIIO interface, since MPIIO deals better with parallel access to a single file than POSIX does.

After some runs of IOR, I realized that something was going wrong internally, since it was yielding abnormally high read speeds. By looking at IOR's source code, I realized that it had some unimplemented functions. Among these functions, the one that affected me was "IOR_Fsync_MPIIO".

One of the input parameters of IOR is "-fsync". By using this parameter, we force the underlying file system to maintain consistency in the local storage by transferring all the information written to memory to the storage device. The problem is that this option is only supported if we use the POSIX.

The equivalent call for MPIIO is MPI_File_sync(). This call should be placed within the method "IOR_Fsync_MPIIO". However, in order to avoid further modifications in the code, I opted for including this call under the function "IOR_Close_MPIIO", so that before closing a file with MPI_File_close(), all the contents would be transferred to disk.

In my configuration, IOR uses a single shared file by all the processes involved in the test. The following figure depicts this situation [8]



The shared file is divided in different segments. Each of these segments is also divided into a number of blocks which have a size that must be a multiple of "transfer size". The processes accessing the file can read as much as "transfer size" bytes at a time and each process is assigned a different block within each segment. Thus, this transfer size corresponds to the actual amount of data transferred from the processor's memory to the file in each I/O function call.

B. S3Bench

In order to benchmark S3, I had to develop my own benchmark suite, since none of the widespread benchmarking tools can be used to test storage like this. To achieve this, I used Amazon's AWS SDK, which provides several methods to write (PUT), read (GET) and delete (DELETE) objects and create/delete buckets.

This program, written in java, covers all the parameter space and returns a file containing all the results.

C. DynamoDBench

Like S3, there is not a widespread benchmark for Amazon's DynamoDB, so I wrote my own benchmark by using Amazon's AWS SDK.

D. EC2Cluster

Configuring a fully working cluster with support for some specific file system may be a tricky task. For this reason, I developed a tool which, along with several scripts, allows users to fully configure and run a fully customizable cluster on top of Amazon EC2 infrastructure.

For my study, I customized this program to easily build a NFS/PVFS cluster with MPI support, so that I could take advantage of MPI to run the benchmark simultaneously on all the clients. These are some of the different parameters accepted by the program:

- Server instance type
- Client instance type
- Availability zone
- Maximum bids for clients/servers
- Security group
- Number of servers/clients
- File system
- MPI process mapping

As well as running a cluster, it allows the user to terminate it.

VIII. Parameter space and Testbed

Defining a plausible parameter space is as important as obtaining the proper results. If we put together all the different instance types in EC2, the different storage options, access patterns and IOR configurations, we may end up with thousands of different tests to be covered. With this in mind, I had to decide what the most important issues were and discard those which would not yield any significant result.

My study can be divided into four different parts: EC2 micro benchmarks, S3, parallel file system benchmarking and finally, DynamoDB benchmarking. A description of each of them is included below:

A. EC2 micro benchmarks

These cover all the different instance types (except for the cluster and micro instances):

- m1.small
- m1.medium
- m1.large
- m1.xlarge
- m2.xlarge
- m2.2xlarge
- m2.4xlarge
- c1.medium
- c1.xlarge
- hi1.4xlarge

For my study, I have used EBS backed instances. As I said before in this document, each of these instances includes some amount of instance store, which usually comes mounted and formatted. If it was not the case, I chose formatted them using ext4.

Besides, I attached an EBS volume to each instance, with a size which varied based on the instance type, since it should be at least twice its memory. If we do not verify this, we will get incorrect read results, because all the contents of the file created by IOR will be read back from memory instead of the actual EBS volume.

With this in mind, in each instance, I had to benchmark both EBS storage and instance store. To observe the influence of the intermediate I/O buffers, I ran IOR with ten different transfer buffer sizes: 4 KB, 16 KB, 64KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 8 MB, and 16 MB. A description of how IOR works can be found in section VIII.A. The block size and segment count were established based, again, on the memory size.

Each test is repeated three times in order to get more fine-grained results.

B. S3

For S3, I considered all the previous instance types and also three different regions (us-west-1, us-west-2, us-standard). In this case, I used my own benchmark, which obtains write/read throughput for different zones and different file sizes: 4 KB, 16 KB, 64KB, 128KB, 512KB, 1MB, 4MB, 16MB, 64MB and 128 MB.

My program automatically creates a bucket in each specified zone and writes/reads to/from that bucket. It also works as a multithreaded client, allowing doing multiple requests from the same instance to the same bucket at the same time. However, this part of the benchmark should be revised in the future to improve the method that I used to compute the aggregate bandwidth among the different threads.

Another test that I found interesting was to request different files from different instances simultaneously to the same bucket. With this, I would know whether Amazon implicitly limits the bandwidth of a bucket or it is unlimited as they claim.

C. NFS and PVFS

After running micro-benchmarks on each instance, it makes sense to see how they behave when working in parallel. Specifically, my objective was to measure the performance of both PVFS and NFS. To benchmark these file systems I used IOR along with MPICH2, allowing me to run the same test simultaneously on all the clients.

For NFS, the cluster size ranged from 2 nodes (1 client and 1 server) up to 65 nodes (64 clients and 1 server). On the other hand, PVFS' cluster size ranged from 1 node to 64 nodes. In the latter, each node acts as a metadata and an I/O server at the same time. I tried to scale these clusters up to 128 nodes without success due to Amazon restrictions affecting the number of running instances/spot requests.

In the NFS cluster, I used IOR with both MPIIO and POSIX APIs and 2 processes per node synchronized with MPI. However, the combination IOR/MPICH2/POSIX was not possible without configuring the PVFS2 kernel interface, which cannot be done with the Linux kernel version that I was using in the instances (it is an amazon specific version). Thus, on the PVFS2 cluster I could only use the MPIIO interface.

Instead of covering all the instance types, I decided to go with the m1.medium, since they have proved to be one of the most cost-efficient in terms of compute capacity, network performance and storage. However, for NFS, I set the server to run on an m1.xlarge instance, because a smaller one would end up being a serious bottleneck for big clusters.

Regarding the storage devices under these file systems; I used instance store (physical attached drive), EBS volumes (the instance volume itself) and also the /dev/shm device which is backed by RAM memory. The latter one has been used to emulate a cluster backed with high I/O devices, such as a cluster composed of hi1.4xlarge instances.

As in the micro benchmarks, I ran IOR with different transfer sizes, block size and segment count to adapt it to each different cluster.

The entire network, MPI and file system configuration has been made through the EC2Cluster tool, which I wrote for this project.

D. DynamoDB

Despite not being a storage solution, DyamoDB has gained importance as one of the most robust alternatives to conventional (SQL-based) relational databases. Its NoSQL nature allows customers to bypass all the problems related with database scaling, management, reliability and performance.

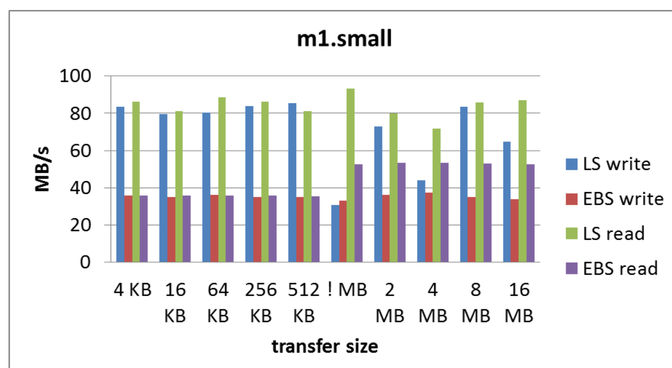
According to Amazon, each data item stored in DynamoDB is automatically replicated across three different availability zones within the same region, providing high availability and data durability. Besides, by using Solid State Drives as storage, the I/O performance of DynamoDB can keep up to the most demanding application in terms of throughput and requests volume.

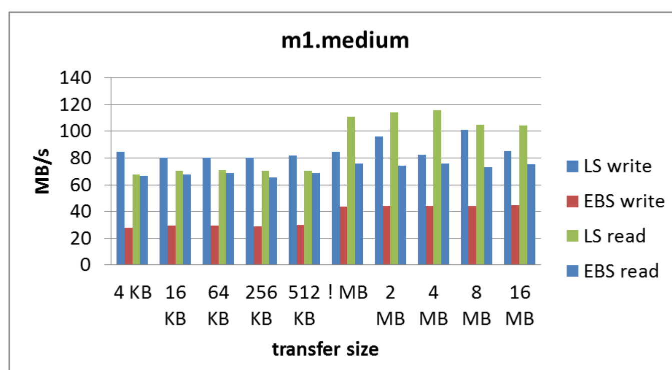
To measure its performance, I wrote a little java program which creates a table, puts a big number of items to the table and then gets all those items back. By doing this and taking into account the item size, I obtain an estimate of both read and write throughput, which might be constrained by the network bandwidth due to the high I/O performance offered by SSDs.

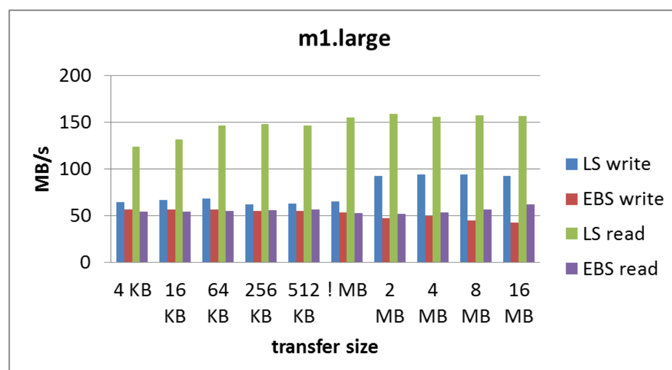
IX. RESULTS

A. EC2 micro-benchmarks

The following charts show the results obtained after running IOR on each of the previously mentioned instances with different transfer sizes and storage devices.

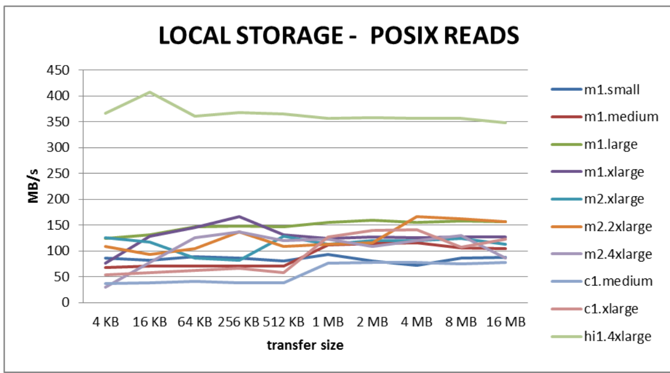


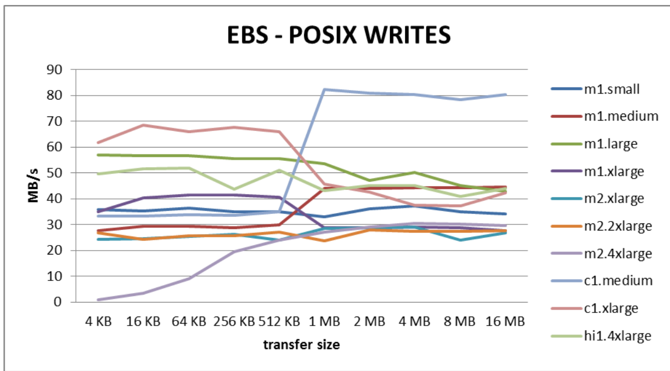


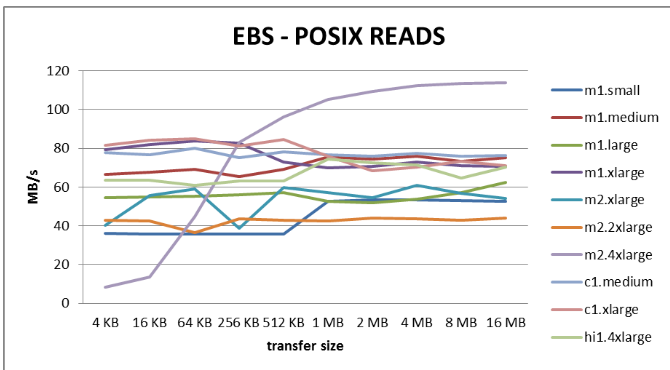


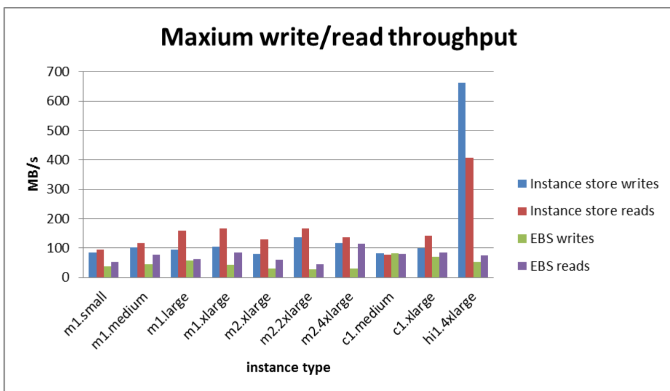
We can see that EBS volumes show a very poor performance, which comes from the fact that they are attached over the network. This fact is noticeable if we carefully examine the all the charts, where we will see that neither write nor read throughput exceeds ~120 MB/s in any of the instances. Taking into account that all the instances are bundled with a 1Gbps Ethernet interface, these results make a lot of sense.

Regarding the instance store, we cannot appreciate a great difference among the instances' throughput. The best performing instances are those with higher memory (m2's) and the overall outperformer is the hi1.4xlarge, since it is equipped with SSDs.







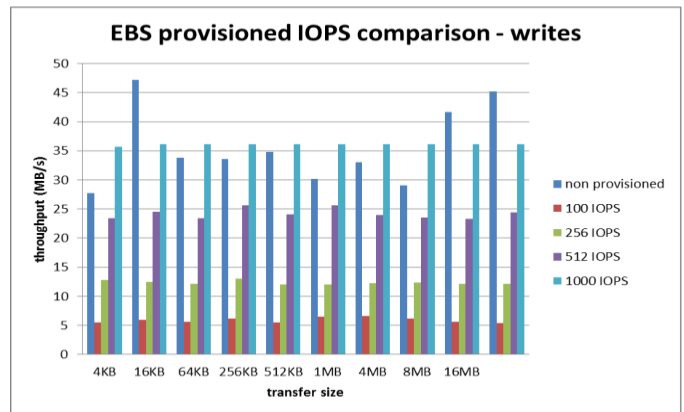


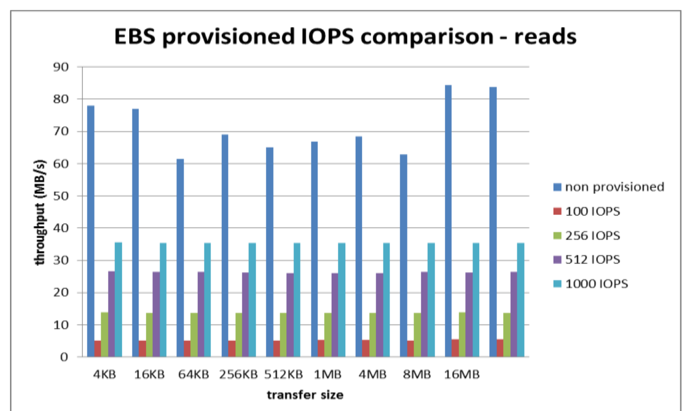
The following table summarizes the previous chart, showing the highest throughput obtained for each of the different studied instances, both in reads and writes and local storage and EBS.

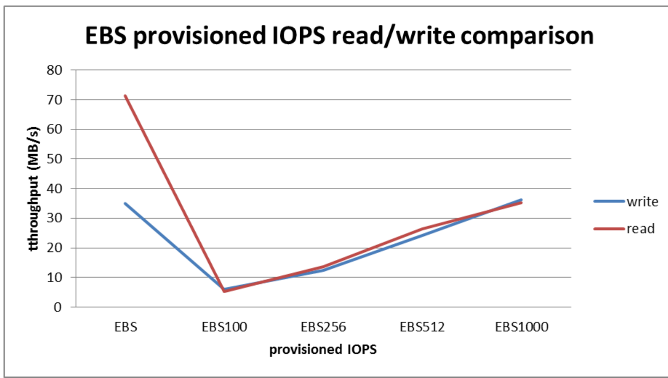
instance	Throughput (MB/s)			
	LS write	*LS read*	*EBS write*	*EBS read*
m1.small	85.31	93.25	37.31	53.29
m1.medium	100.95	115.55	44.52	75.78
m1.large	94.63	159.43	56.82	62.31
m1.xlarge	104.50	166.12	41.50	83.75
m2.xlarge	78.75	127.97	28.91	60.81
m2.2xlarge	136.81	166.64	27.82	43.96
m2.4xlarge	117.20	136.74	30.43	113.81
c1.medium	81.63	77.15	82.40	80.20
c1.xlarge	98.94	140.32	68.39	84.88
hi1.xlarge	661.38	407.64	51.83	74.40

Recently, AWS announced a new feature called “EBS with provisioned IOPS” [9]. These are EBS volumes for which they guarantee a given amount of IOPS, which is specified by the client during the creation of these volumes. The amount of allowed IOPS ranges from 100 to 1000 and is limited by the actual size of the volume by a proportion of 10 to 1 (if your volume is 10 GB, you can choose 100 IOPS at maximum).

The following graphs contain the results from running the previous benchmark in different IOPS provisioned EBS volumes. All of them have been run in a c1.medium instance.



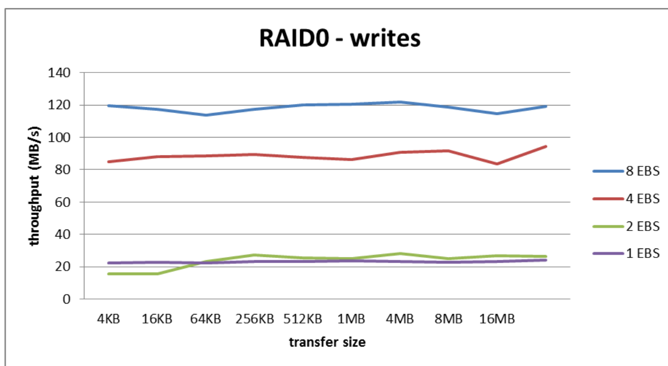


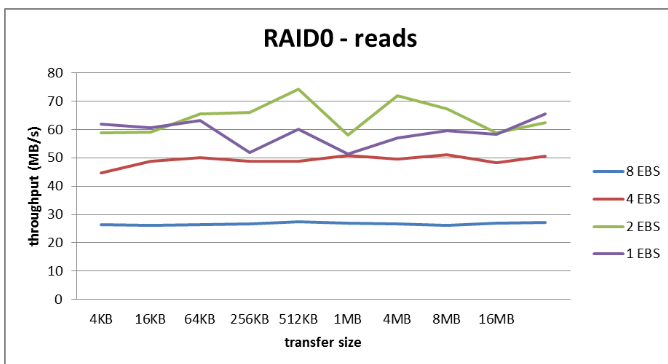


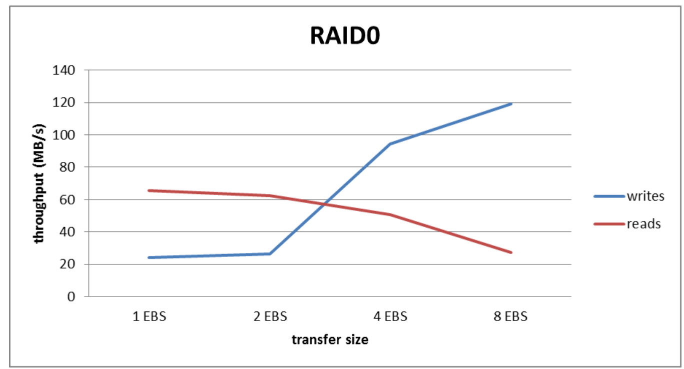
For this benchmark I used four different IOPS provisioned EBS volumes and one standard EBS volumes. Surprisingly, the standard EBS volume outperformed the others in all the tests.

However, the provisioned IOPS EBS volumes seem to behave very well in terms of stability, since they provide the exact same throughput in all the different tests, in spite of the different transfer size, which seems to slightly affect the standard EBS volume.

Finally, to complete these micro-benchmarks, I set up a software RAID-0 with EBS volumes, varying the number of volumes from 1 to 8. I ran the same benchmark on a c1.medium instance and these were the results:





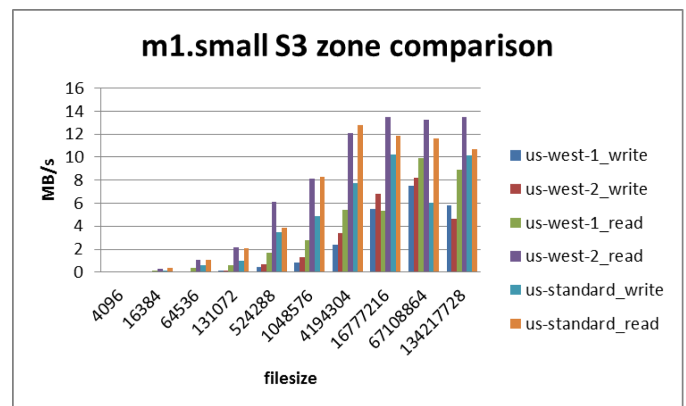


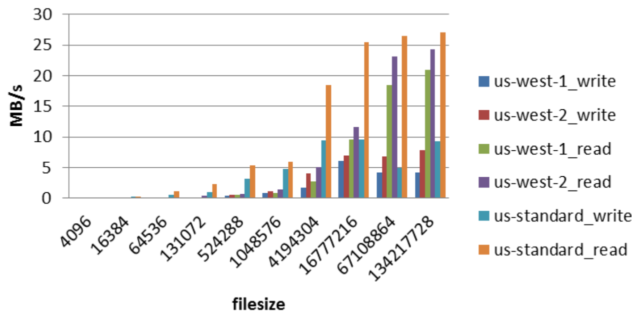
Here we see how the write throughput increases with the size of the RAID and on the other hand, the read throughput does exactly the contrary. However, both of them keep nearly constant as we vary the trasnfer size and the maximum achievable throughput is around 120MB/s, which is the maximum network bandwidth for this type of instance.

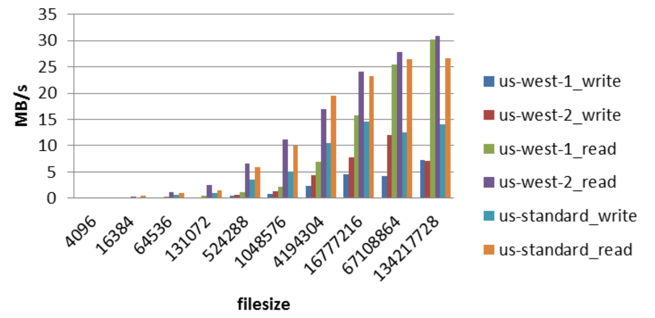
B. S3

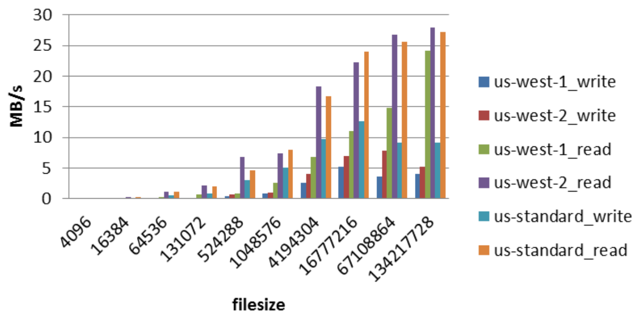
We can find up to three different regions in the USA where you can create a storage bucket for S3. Internally, these regions are known as us_west-1, us_west-2 and us_standard.
Choosing one region or another depends on the latency and the type of consistency that we want for our data. Whereas the us_standard region provides eventual consistency for all requests, us_west-2/1 regions assure read-after-write consistency for PUTS of new objects in a bucket and eventual consistency for overwrite PUTS and DELETES.
All the instances, except for the hi1.4xlarge were running on us_west-2. For this study, I have measured the read/write throughput from all the instance types to one bucket in each different region. The following charts show the difference in terms of write/read throughput for each instance between the different regions.

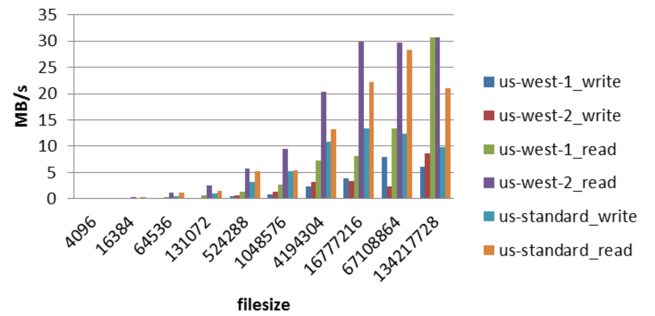
Us_standard region covers both facilities in Northern Virginia and Pacific Northwest. The final destination of the data uplaoded to this region is established by using network maps.

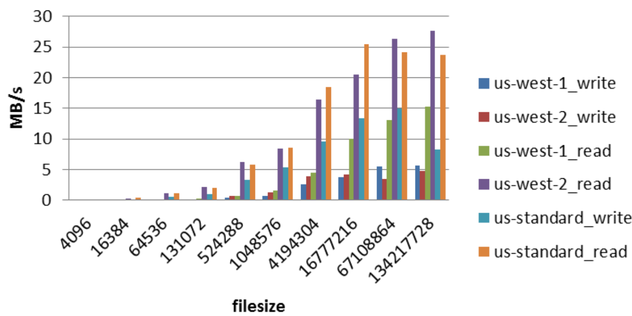


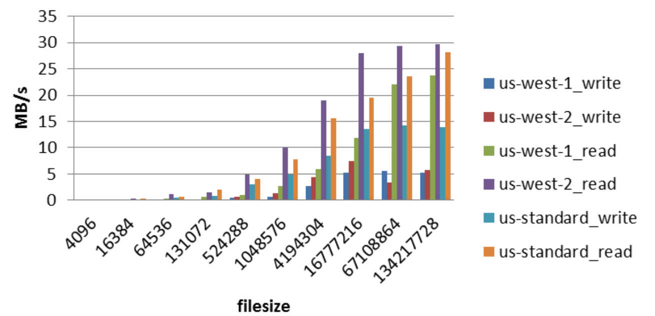

m1.medium S3 zone comparison
MB/s
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read

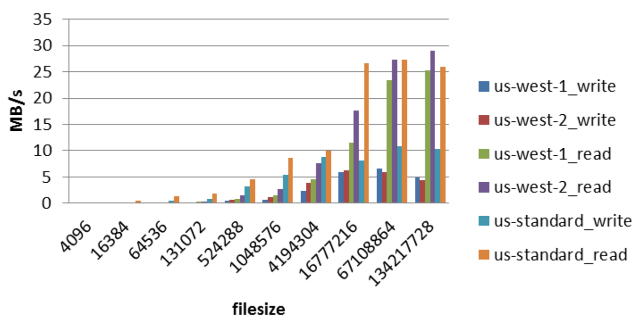

m2.2xlarge S3 zone comparison
MB/s
35
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read

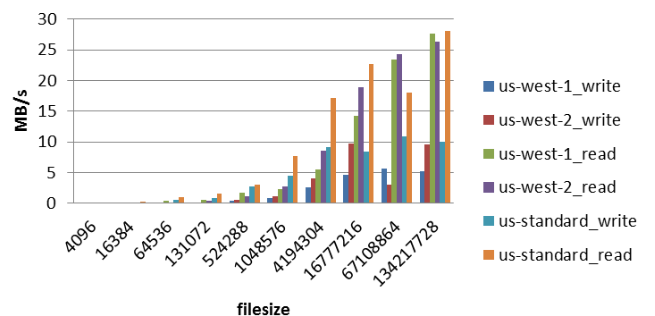

m1.large S3 zone comparison
MB/s
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read

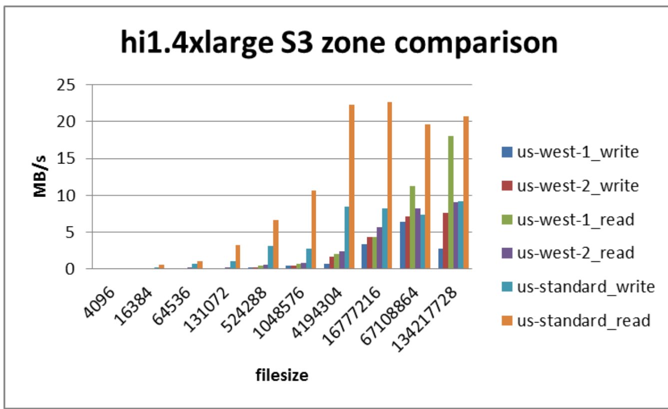

m2.4xlarge S3 zone comparison
MB/s
35
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read


m1.xlarge S3 zone comparison
MB/s
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read


c1.medium S3 zone comparison
MB/s
35
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read

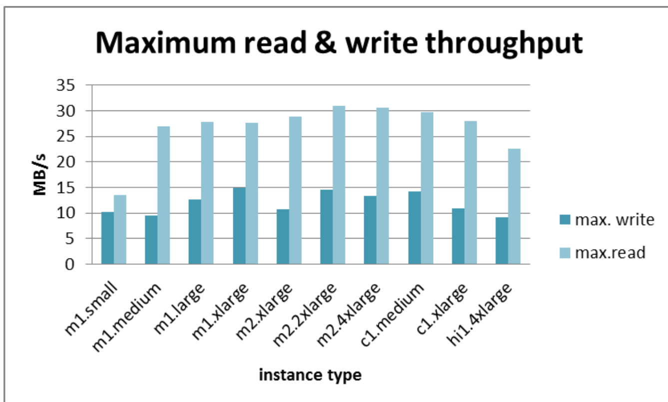

m2.xlarge S3 zone comparison
MB/s
35
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read


c1.xlarge S3 zone comparison
MB/s
30
25
20
15
10
5
0
4096
16384
64536
131072
524288
1048576
4194304
16777216
67108864
134217728
filesize
us-west-1_write
us-west-2_write
us-west-1_read
us-west-2_read
us-standard_write
us-standard_read



Leaving aside the small instances, we cannot appreciate any big difference between the maximum read/write throughput across instances. The reason is that these values are implictily limited by either the network capabilities or S3 itself. Unlike EBS volumes, the links between the data which goes from the instances to S3 might be treated differently in terms of network routing, so that the underlying physical links have lower capacity than those between instances and EBS volumes.

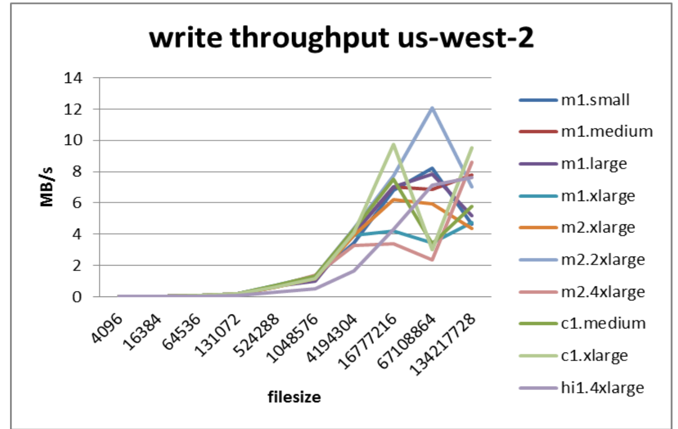
However, since S3 is held back by standard-storage devices (i.e. commodity hardware), the additional redundancy and high consistency provided by S3 may struggle the overall I/O performance offered by this service.



instance	Throughput (MB/s)	
	LS write	*LS read*
m1.small	10.26	13.49
m1.medium	9.53	26.99
m1.large	12.66	27.85
m1.xlarge	14.93	27.57
m2.xlarge	10.79	28.92
m2.2xlarge	14.59	30.85
m2.4xlarge	13.37	30.66
c1.medium	14.17	29.63
c1.xlarge	10.91	28.05

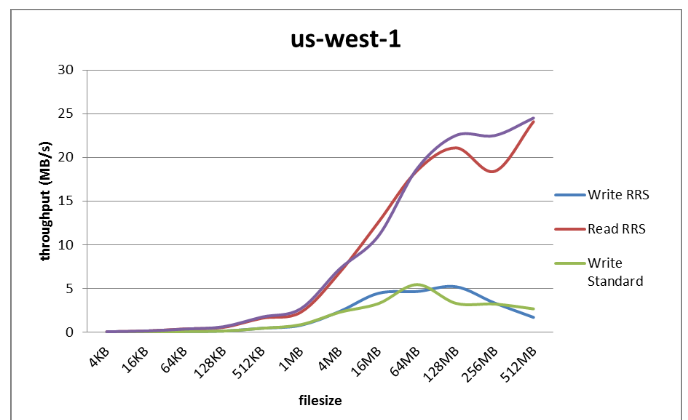
instance	Throughput (MB/s)	
	LS write	*LS read*
hi1.xlarge	9.14	22.65

The following graph shows that the maximum write throughput saturates at a filesize of 16 MB. Increasing the filesize does not make any significant difference from that point.

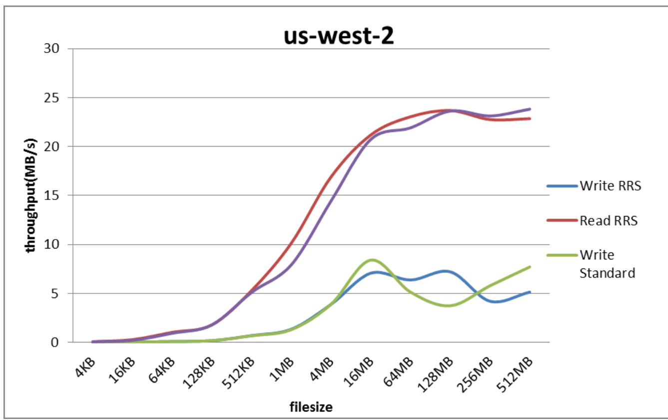


AWS offers its users the option to store their data in S3 at a lower cost by providing less redundancy than Amazon's S3 standard storage. They call this feature Reduced Redundancy Storage (RRS). Since data is replicated fewer times, the cost is lower. However, I wanted to know whether this had some effect on other aspects regarding performance, such as write and read throughput.

To do this, I ran a S3Bench on a m1.medium instance on us-west-2 and checked whether there was any difference when uploading and downloading files with RRS. The results are shown below:



In this zone, we can see that both writing with RRS enabled and RRS disabled is affected by the filesize, since write throughput slighly decreases as we make the filesize bigger. This is due to the latency between regions us-west-1 and us-west-2.



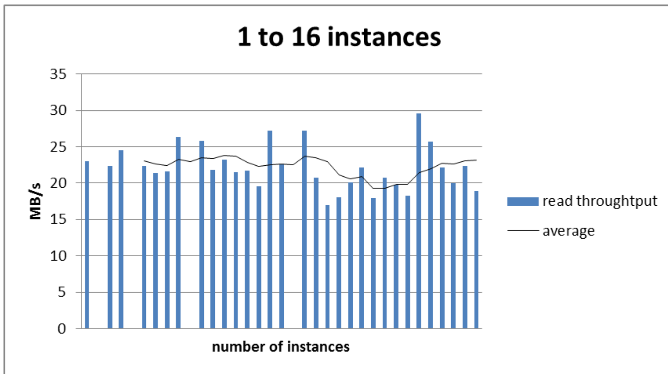
On this region, we cannot appreciate much degradation in the write throughput as we increase the filesize. In this case, both the running instance and the storage are within the same region.

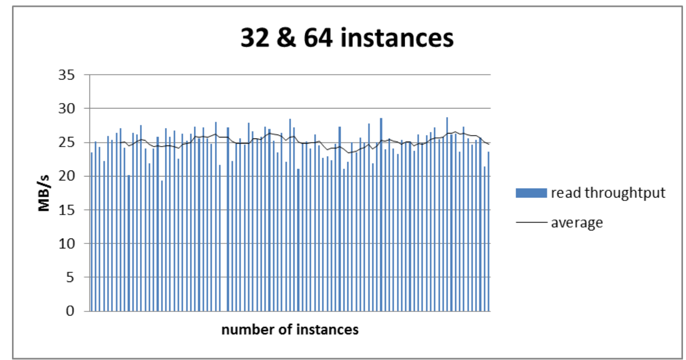
Regarding the RRS option, we can see that there is not any noticeable difference between uploading objects with this option or leaving it as is. The slight changes that we see are due to changes in the network load at each time.

Finally, I wanted to check whether Amazon has some type of bandwidth allocation policy by which several different instances accessing the same bucket at the same time would see their individual bandwidth affected. I used the tool S3Bench along with EC2Cluster and some parsing scripts to launch all the needed instances and run the benchmark at the same time. The results obtained from this tests are divided in reads and writes and can be seen below:

- reads

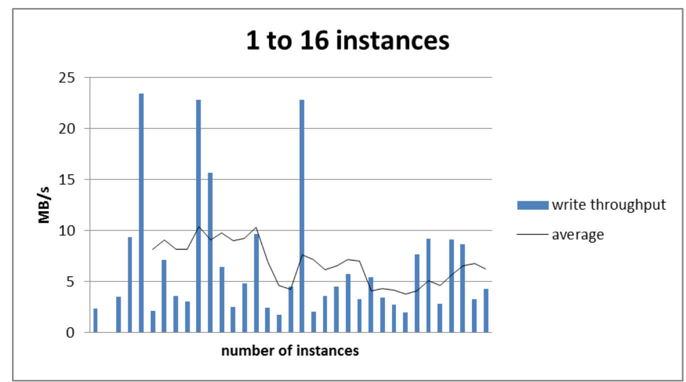
The moving average does not show any trend that would indicate a lower individual throughput as we increase the number of simultaneous instances.

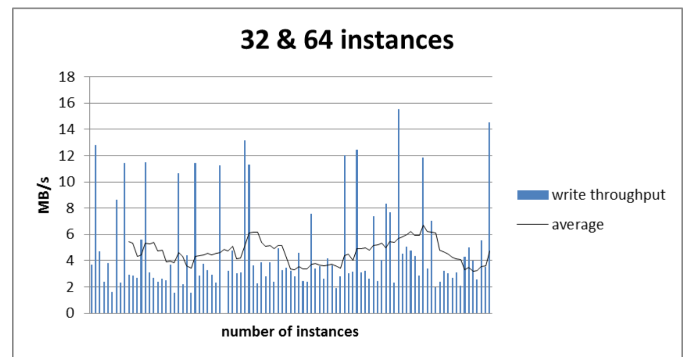




- writes

Despite not being very stable, write operations do not seem to be affected by any kind of bandwidth allocation policy. The peaks that we observe may be due to the status of the network link between the instances and S3 servers.





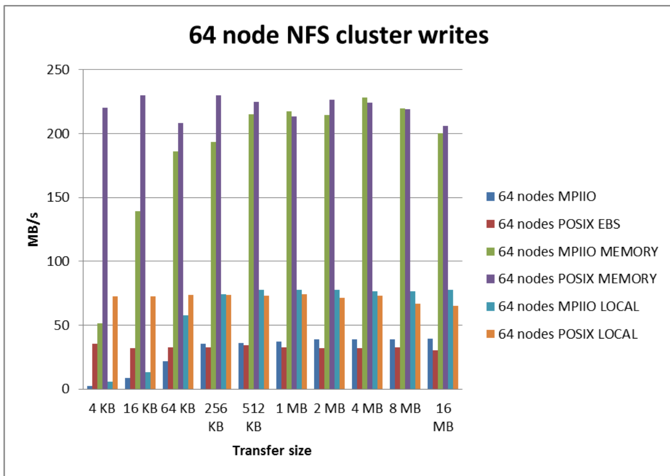
.The conclusion that we can draw from these results is that S3 does offer a quality service in terms of bandwidth allocation, altough its overall throughput may be very low for applications requiring fast I/O access to data, as we observed in the individual instance study.

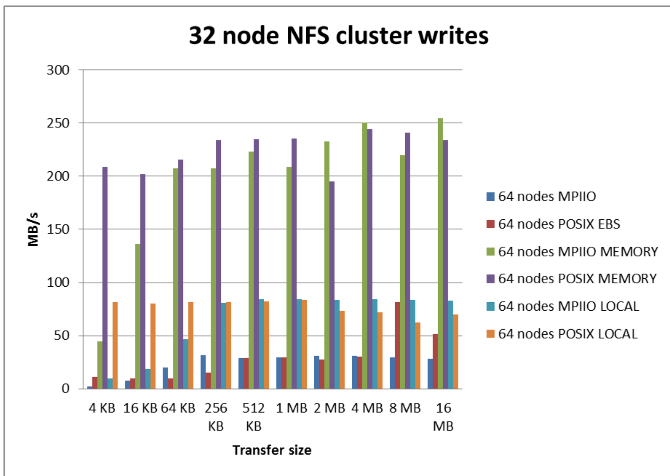
C. NFS

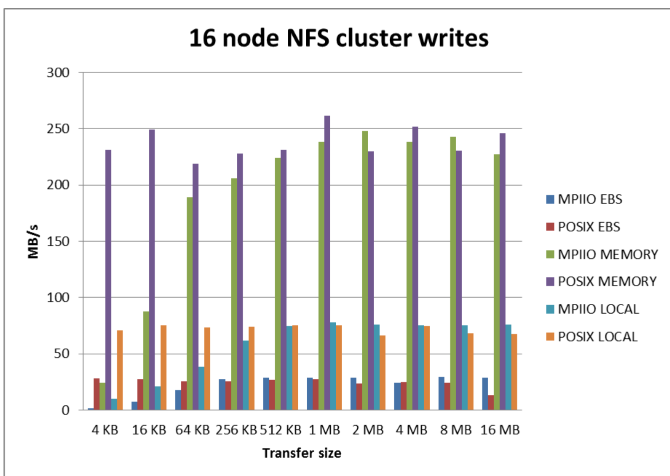
Its centralized topology does not make NFS the most suitable file system for the applications which may be of interest for readers of this document. However, NFS is still being used in many scenarios where a centralized source of information is needed.

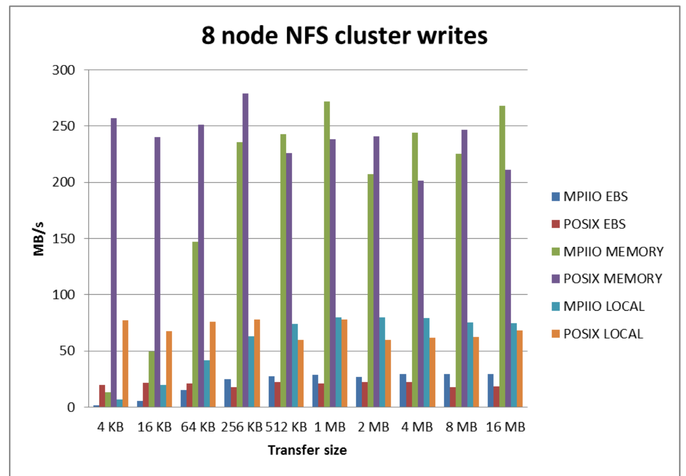
In these tests, I simulated some of those scenarios by setting up clusters of different sizes, varying the storage devices, basic I/O transaction size and access interface.

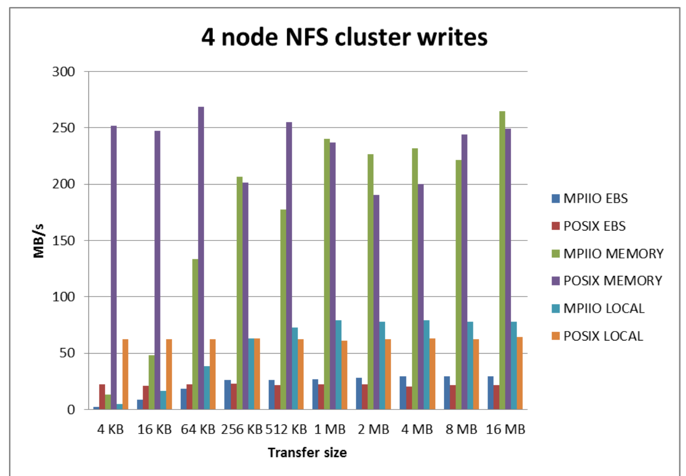
- writes

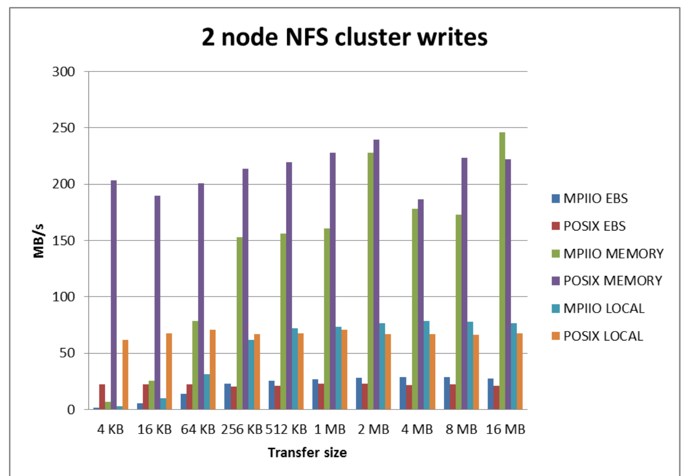


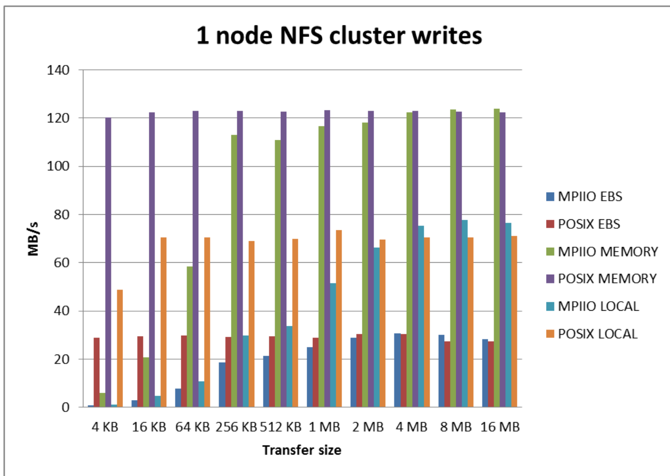










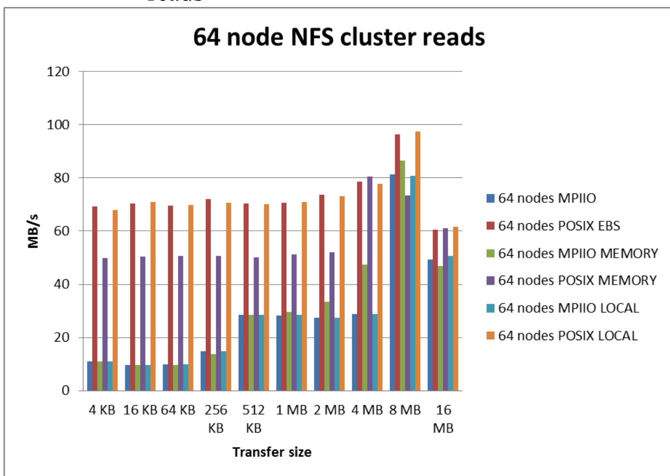


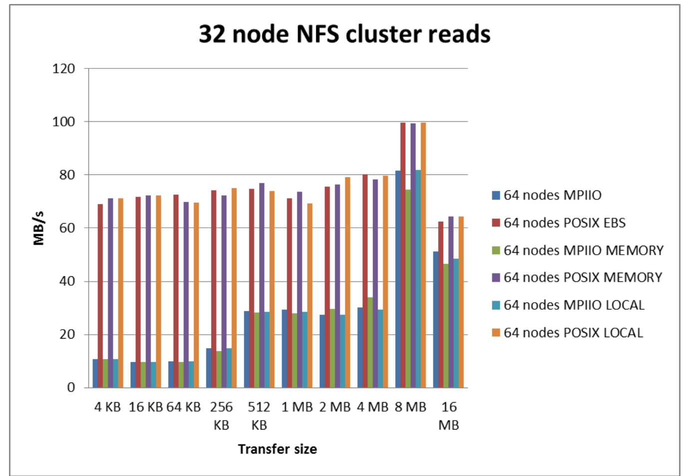
In these charts we can observe different things. First of all, memory, as expected, yields much higher throughput than EBS and instance store altogether. However, whereas POSIX writes keep almost constant across the different tests, MPIIO tests suffer a great performance improvement as we increase the transfer size. This size is directly connected with the number of basic I/O transactions and hence, the overhead associated with each of those transactions. Therefore, by increasing the transfer size, we reduce the number of transactions and thus, the additional overhead introduced by MPIIO. This makes the throughput noticeably higher.

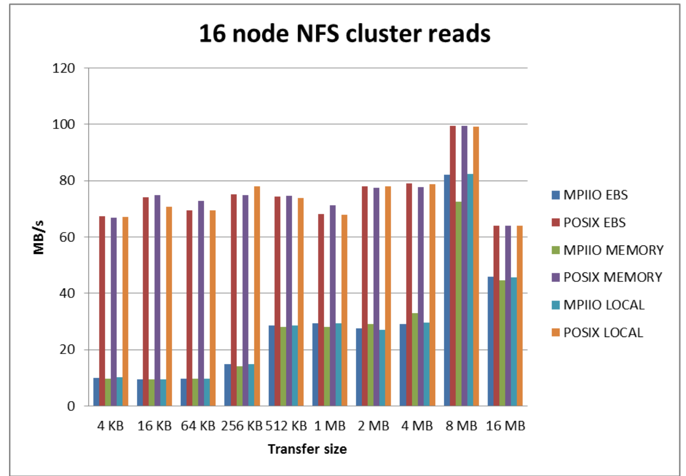
Another remarkable fact is that the aggregated write throughput across all memory tests except for the 1 client/1 server benchmark is higher than the maximum theoretical network bandwidth for a m1.xlarge instance, which has been the one used for the server in all cases. This may mean that the 1Gbps interface is virtualized on top of a higher capacity one, and Amazon does some kind of internal traffic differenciation which allows higher throughput for some data flows.

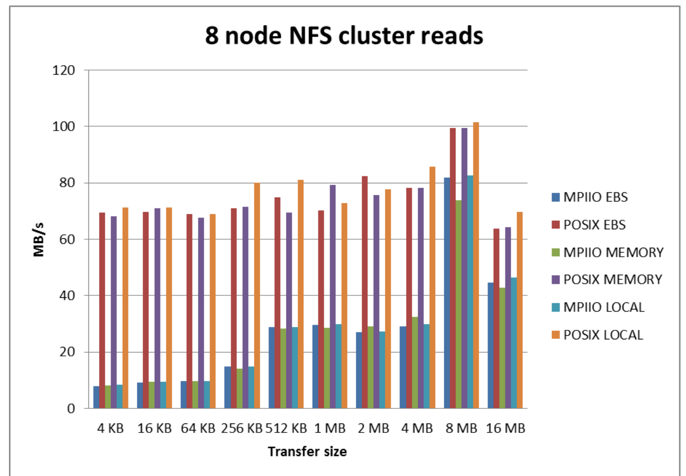
In contrast with it, we can see that the 1 client / 1 server configuration does not yield any throuhgput higher than ~120MB/s, which is congruent with the maximum network capacity of the server.

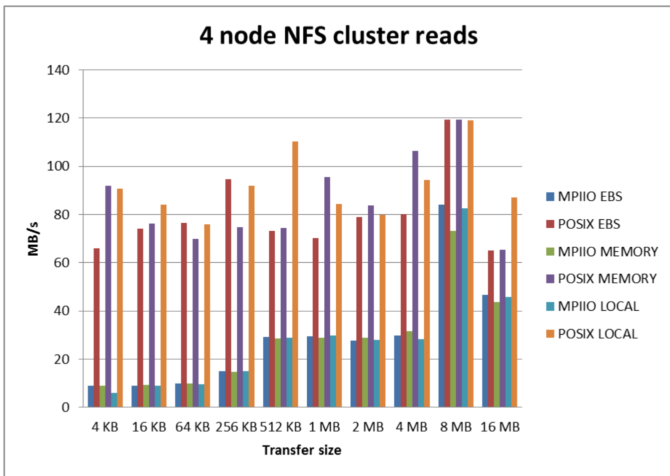
- reads



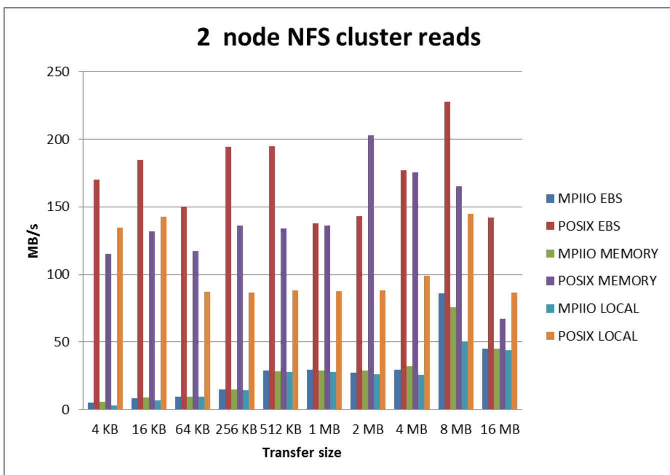


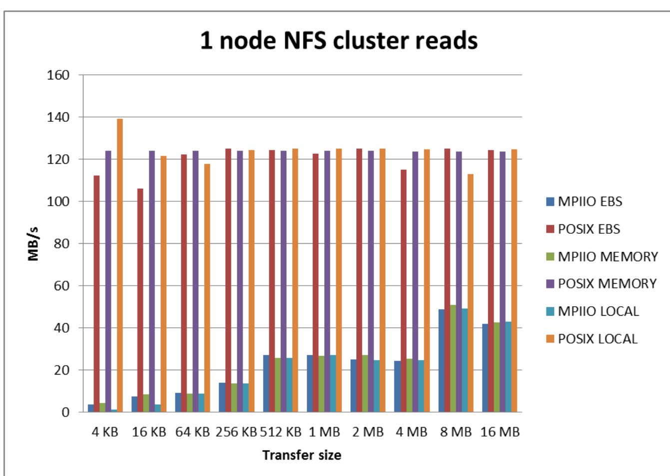




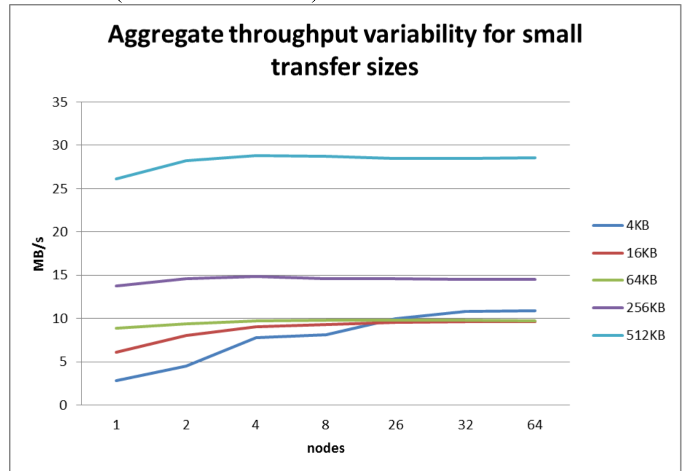


The benchmarks up to 4 clients / 1 server seem to follow the same pattern in terms of aggregate throughput across clients, with very similar results. However, in this case, none of them exceeds the maximum theoretical bandwidth allocated for a m1.xlarge instance (1 Gbps).
In this case, it seems that Amazon does not allow higher bandwidth allocation for incoming data flows.





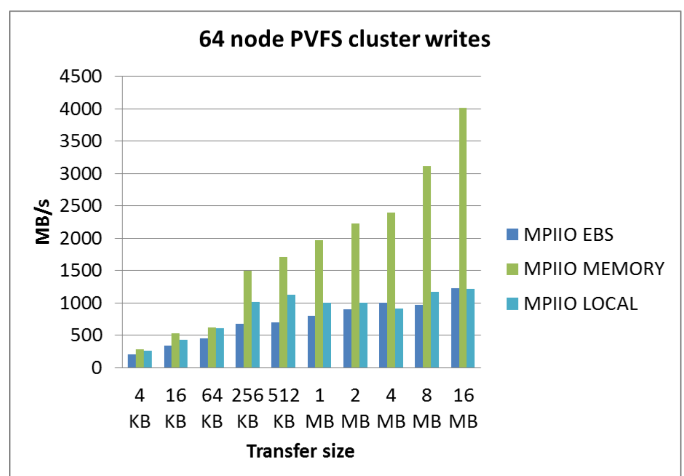
The following graph shows the variability of the aggregate throughput across clients when using small transfer sizes (<1MB). The data plotted here have been obtained from averaging the MPIIO results, which are quite similar within the same test (same transfer size).

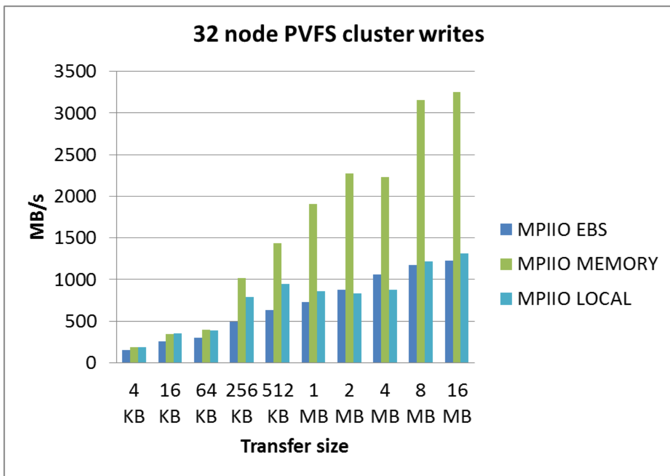


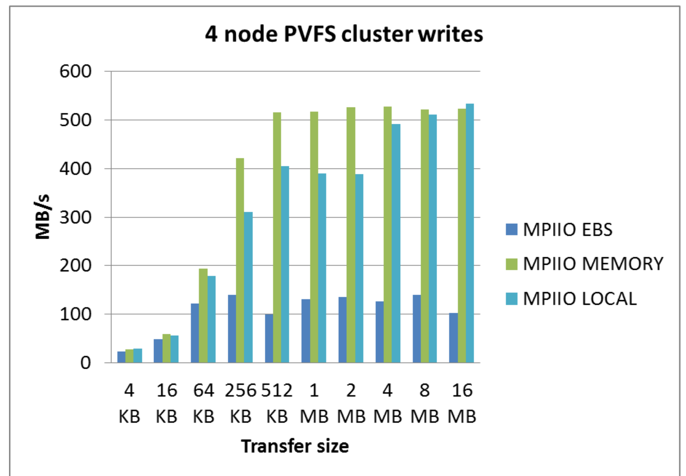
As wee see, the greater the transfer size, the more plain the lines are. This depicts how the overall aggregate throughput is less sensitive to changes in the size of the cluster as we increase the basic I/O transaction size.

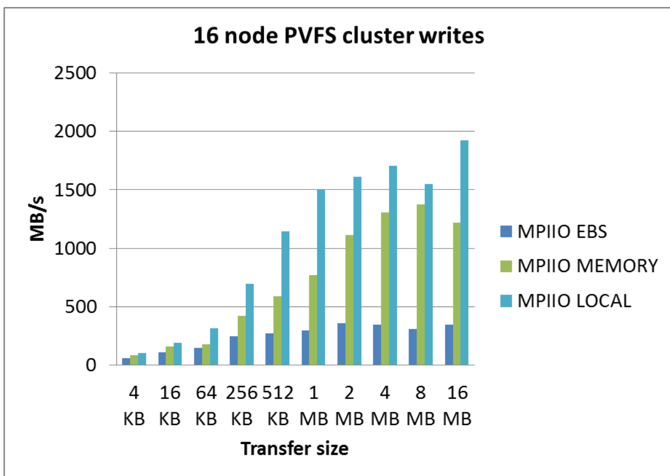
D. PVFS2

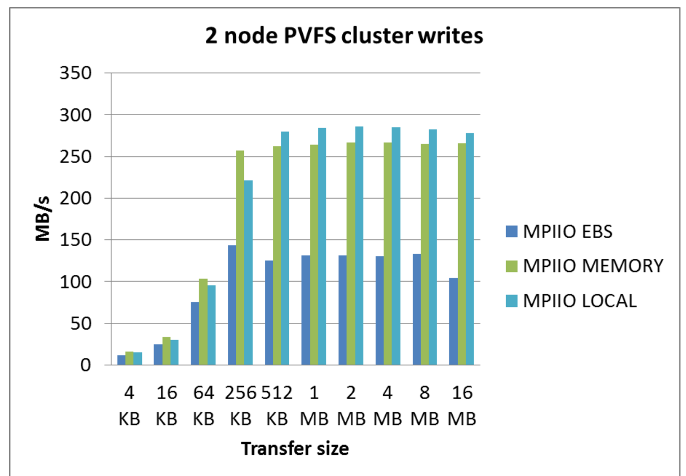
To benchmark PVFS2 I only used the MPIIO interface, since MPICH2 implicitly includes support for PVFS2. Unlike NFS, there is not a centralized source of data. Here, each file is divided into different parts, which are stored in those nodes working as I/O servers. The metadata associated with these files is stored by the Metadata servers. In the configuration that I used, every node in the cluster serves both as an I/O and Metadata server.
The main problem of this configuration is that having such an amount of metadata servers increases the network overhead to a great extent. As an advantage, it drastically decreases the probability of the whole file system to go down, since it does not have a single point of failure, as NFS does.

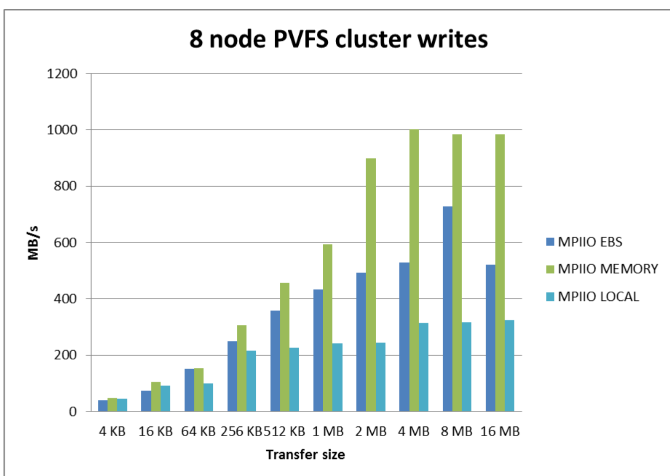


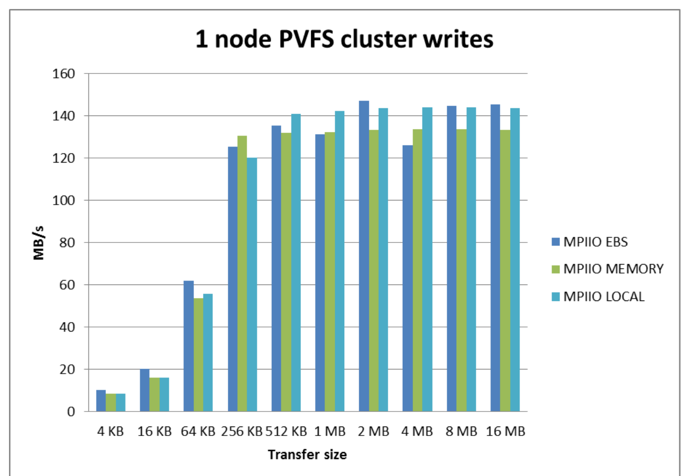






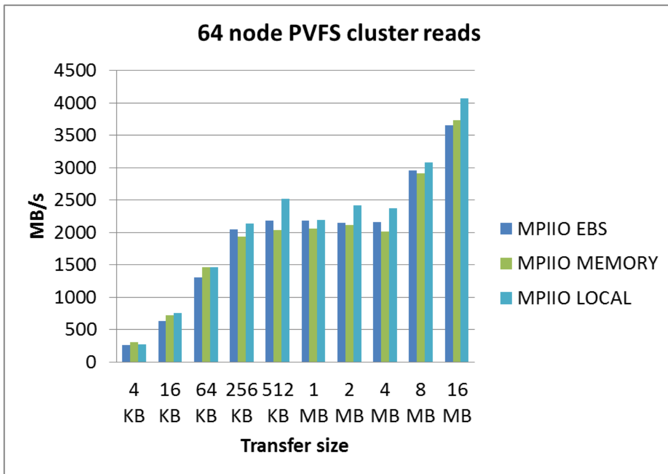


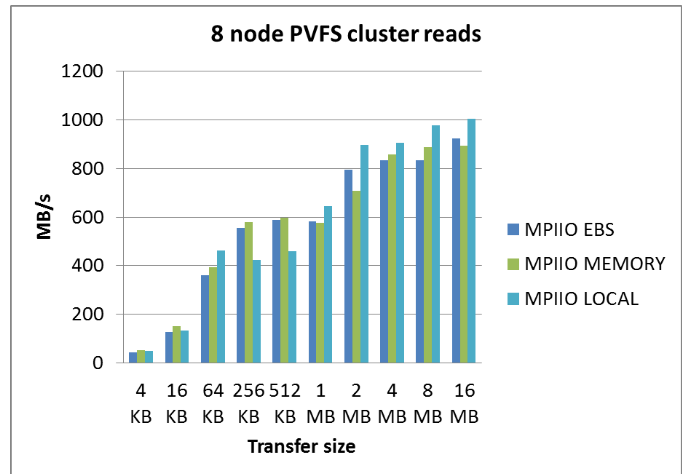


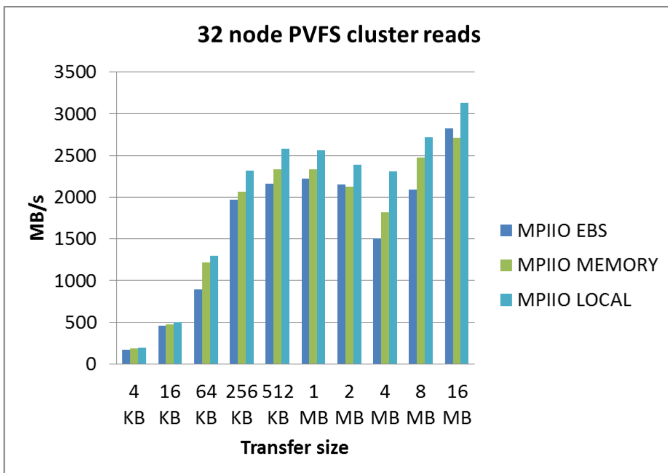


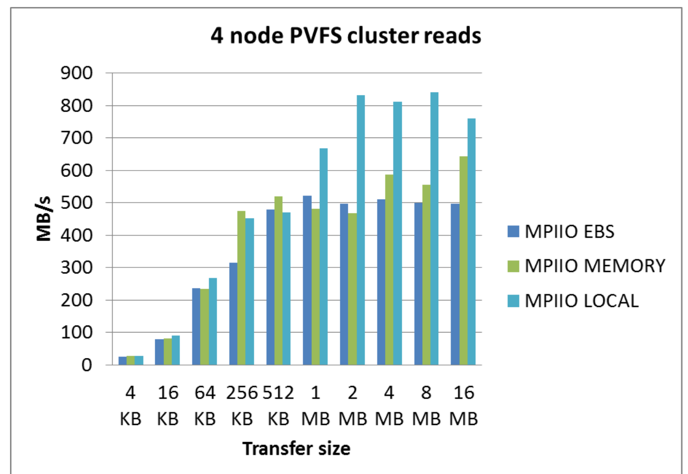
- As expected, the memory-backed pvfs2 cluster outperforms the rest in most of the tests. However, since in this case the throughput is not limited by the maximum network capacity of a single instance, but (theoretically) by the aggregated maximum network capacity, the instance-store-backed cluster performs really well in comparison with EBS and in some cases, memory.

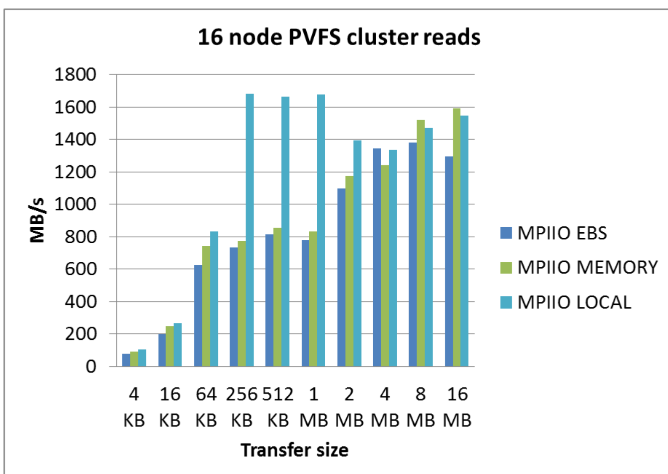
- reads

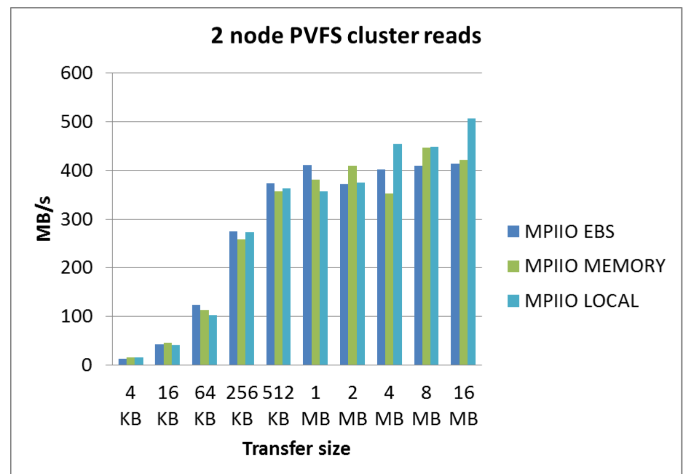


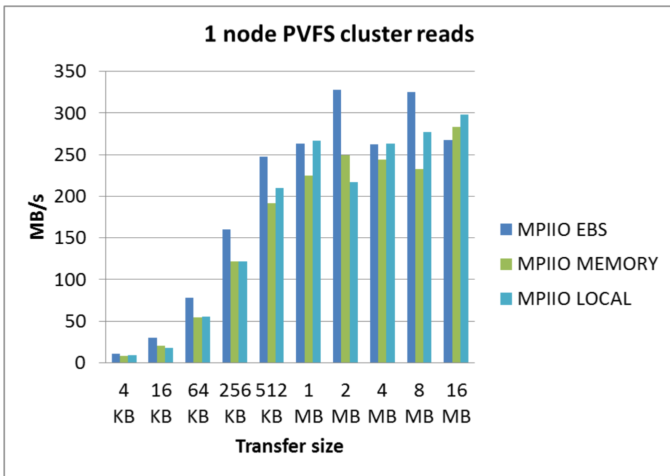


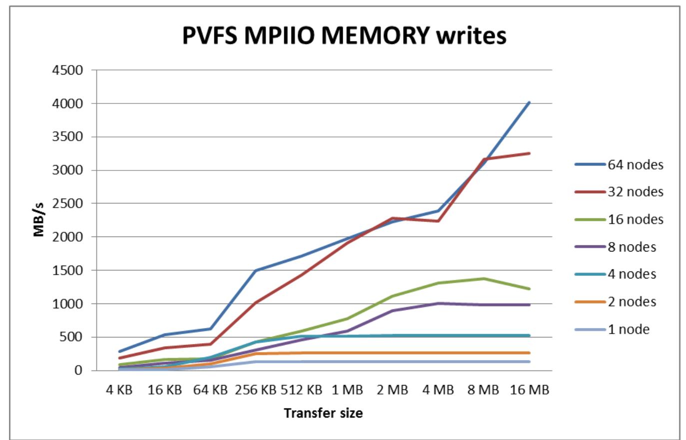




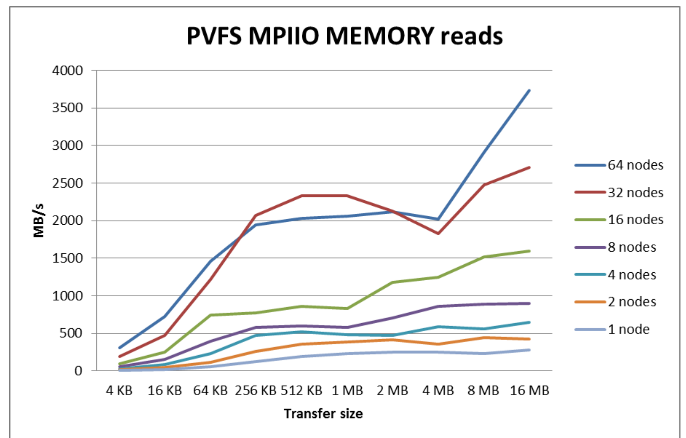


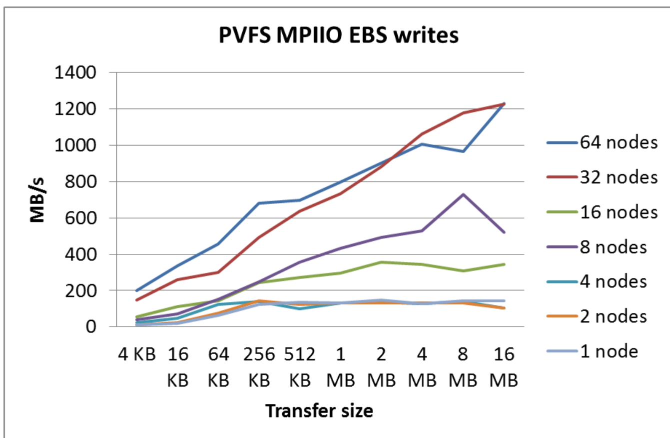


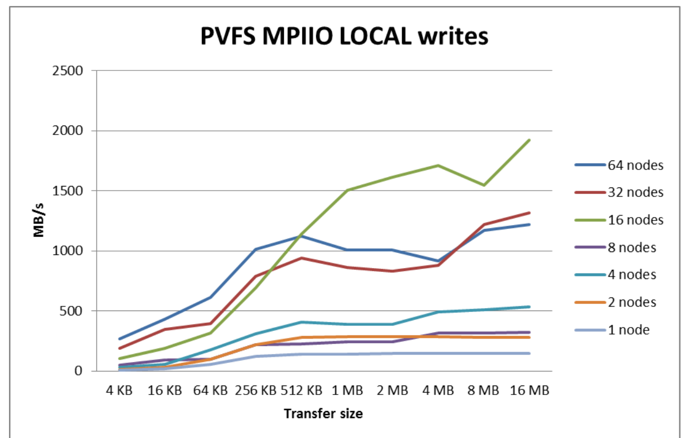


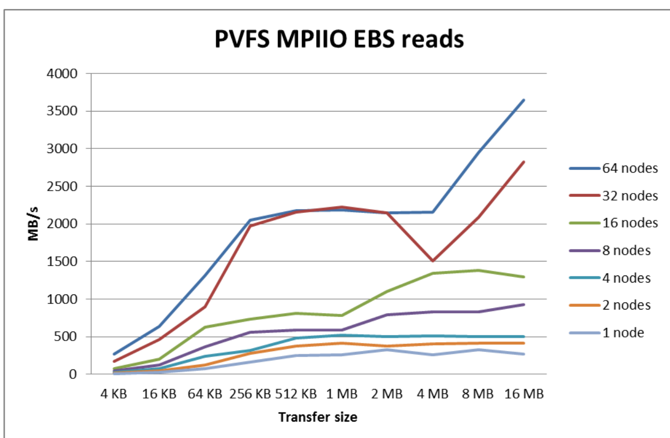


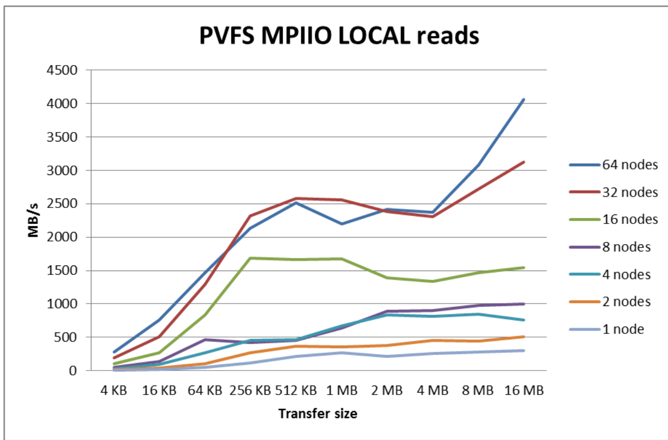
In general, the effect of having a small transfer size is reflected in all the graphs, where we see that the throughput inceases as we make the transfer size bigger. Again, this fact is due to the overhead added by each little I/O transaction.
To have a more detailed view of the behaviour of PVFS2 on different storage options, I have included the following graphs:

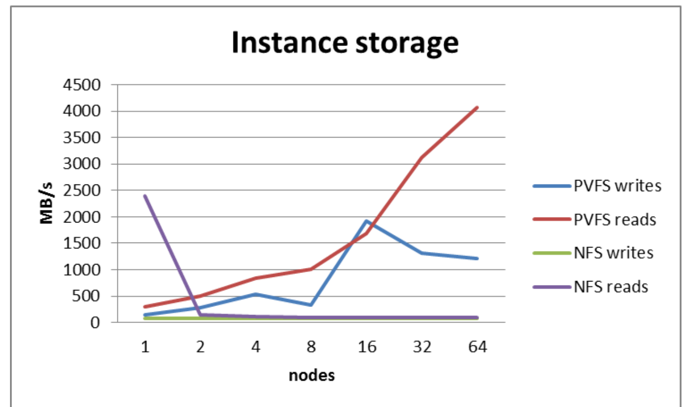








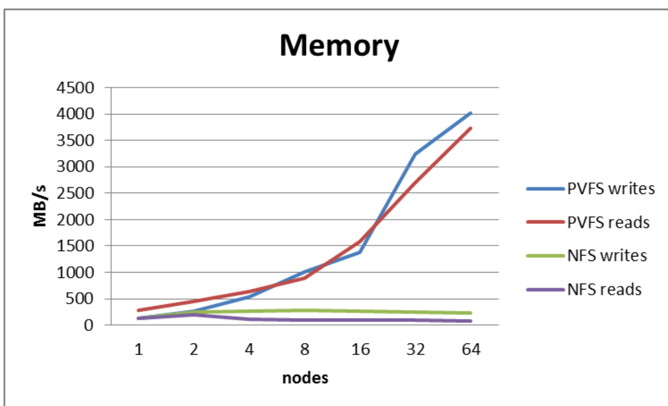


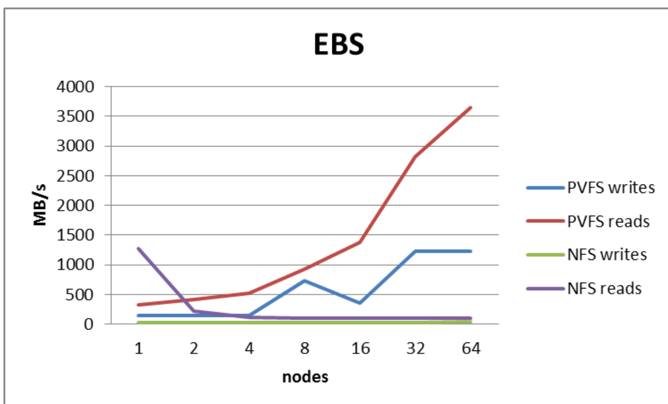


We can observe how write/read throughput is affected by the basic I/O transaction size in each different storage option and cluster size. For example, we can see how the write throughput tends to increase more linearly with the transfer size as we make the cluster bigger.

Reads, on the other hand, show a more moderate growth in all cases, although it also tends to increase linearly in big clusters.

Finally, the next graphs serve as a summary to show how the cluster size affects overall read/write throughput using different storage options, for both NFS and PVFS2. For these graphs I have used the highest values obtained from the benchmarks for both PVFS2 and NFS.





X. CONCLUSIONS

Amazon's cloud, due to its size and versatility in terms of offered services and solutions may be the start point for all of those who think that cloud computing may be displacing some HPC systems in the near future.

Despite being initially thought for web services, AWS is increasingly beginning to be understood as a good alternative to run short term experiments and simulations.

However, we have seen that the performance of some of their services is not what we expected or we would like to be getting from them. Amazon is currently more worried about providing a quality service in terms of security, consistency and durability rather than offering ultra-fast systems capable of outperforming any other machine out there at a fair price.

This trend is said to be changing within a short period of time. Nowadays, more and more web applications require an amount of capacity which is getting higher by the day, due to the nearly exponential growth of their users. The spread of smartphones, bundled with high-speed network access, forces these web applications to be prepared for rapidly changing demand of capacity and thus, resources. For this reason, lots of them rely on AWS to provide a quality service which is unaffected by peaks of activity, since they will be leveraged to the AWS infrastructure.

Amazon is aware of this situation and it is slowly introducing new services and products into their platform which, far from being something really innovative, improves the existing ones. In this regard, increasing compute capacity and I/O performance are two of the most important objectives being pursued by Amazon. An example is the introduction of new high I/O instances, equipped with SSDs, provisioned EBS volumes which provide a constant IOPS rate and many more.

The scientific community must keep an eye on the changes occurring in platforms like Amazon's cloud since lots of web applications are beginning to be held back by systems which could also be suitable for the workloads imposed by any scientific simulation or experiment.

XI. FUTURE WORK

The next steps in this study include benchmarking other distributed file systems such as GLuster, Ceph and the under-development FusionFS. With the tools I have developed, it

should be fairly easy to deploy big clusters in Amazon using any of the available configurations.

Regarding Amazon services, another step to be done should be to benchmark cluster instances, which have not been included in this study. These instances provide very high network performance and can be a good starting point to form a cluster out of few resources.

To complete this paper, an economic analysis and a comparison with other available platforms would complement this study.

REFERENCES

[1] "MPI-IO Optimization for Solid State Drives", Pedro Álvarez Tabío-Togores, Jesús Hernández, unpublished.

[2] http://aws.amazon.com/ec2/

[3] http://www.pvfs.org

[4] Gropp, W. 2002. *MPICH2: A new start for MPI implementations.*

[5] http://code.google.com/p/bonnie-64/

[6] http://sourceforge.net/projects/hdparm/

[7] http://sourceforge.net/projects/ior-sio/

[8] "Using IOR to Analyze the I/O performance for HPC Platforms", Hongzhang Shan, John Shalf, National Energy Research Scientific Computing Center (NERSC), Future Technology Group, Computational Research Division, Lawrence Berkeley National Laboratory

[9] http://aws.amazon.com/about-aws/whats-new/2012/07/31/announcing-provisioned-iops-for-amazon-ebs/