Measuring the IO Overhead of Container Technology (Docker)

Submitted By

Ankur Saikia (A20445640)

Guided by

Jaime Cernuda Garcia

INDEX

Abstract - **The rise of containerization technology with systems like Docker, has been a massive revolution for cloud-based systems. Yet, in more performance-oriented systems, such as the HPC community, containers are often left unused due to the well documented slowdowns in processing capabilities due to virtualization overheads. Additionally, there are many container specific configurations that might affect the I/O performance.**

In this project we study the effects of containerization on I/O performance using industry standard benchmarks like STREAM and HACC IO also considering the various possible storage configurations for Docker. We are specifically interested in exploring the memory and storage performance, taking bare metal performance as the base.

From this project we hypothesize that certain Docker configurations would allow us to ameliorate the I/O overhead as compared to other configurations.

1. INTRODUCTION

High Performance Computing (HPC) applications have a high resource utilization (CPU, Memory, IO, throughput …) and hence are traditionally run directly on interconnected bare-metal physical machines. These interconnected machines form clusters and each of them need specialized software to run.

In such scenarios, we have the advantage of the best performance, but there is the disadvantage of setting up and maintaining each node in the cluster. Even though operating systems and dependent applications can be installed using automated scripts, it takes a lot of time. It is also harder to maintain consistency across all the nodes.

Additionally, HPC workloads may face a situations when there is a spike in the data processing requirements and it should scale without any disruption. [1]

Containers are one of those technologies which can be used to tackle both the above problems. They generally have lower overheads than Virtual Machines and are extensively used in the cloud computing industry.

Docker is one such container technology being used in the industry. Docker images can be used to standardize the software installation and dependency management for HPC clusters. Additionally, using Docker container management tools such as Kubernetes, automated scaling and fault tolerance can be achieved.

While this looks all good, Docker itself has its limitations in the fact that it gives root access to the running operating system image which can be used to operate on the host file-system. The same docker daemon can also be used to create new docker containers to gain root access through the Docker daemon attack surface. [10]

In this project we don't consider the security aspect of Docker since there is already an experimental rootless mode [11] in the works, but rather consider the IO overhead of Docker when compared to a bare metal system. We are specifically interested in measuring the memory and storage IO overhead.

Our goal is to compare Docker containers' IO performance using their different storage configurations with the bare metal performance using industry standard benchmarks such as STREAM, HACC IO and Redis-Benchmark.

We make the following contributions:

We provide an up to date comparison of native and Docker environments using recent hardware and software across a cross-section of benchmarks and workloads relevant in the HPC space, additionally comparing different persistent storage options in Docker.

We elaborate on ideas as to how we can decrease the overhead of Docker containers in a real world environment.

2. BACKGROUND

A. Motivation and problem statement

The HPC community doesn't use Docker containers extensively due to slowdowns in processing capabilities because of virtualization overheads. There is also the problem of security in these containers. Hence other container projects specifically designed for HPC workloads like Singularity, Shifter, Charliecloud have cropped up.

This however doesn't mean that we should squash Docker completely. That is because Docker has a much bigger developer base and is also one of the oldest among application container solutions. The current current experimental version of Docker also supports rootless mode which is also an important step in the right direction and we hope to see much better reception of docker in the coming years by the HPC community.

The studies found online are mostly based on old hardware (HDDs) and older versions of both Linux and Docker [2, 3]. The benchmarks used are also outdated and far better industry-centric benchmarks have evolved over the years. We should try to use them and benchmark Docker IO on the latest hardware.

Considering the recent improvements in both hardware and the software components we hope to find less overhead and hence better results than previously found.

B. Linux Containers - Docker

Linux Containers work by modifying the existing OS to isolate processes rather than running a full OS on virtual hardware. They use a feature called kernel namespaces to create isolated containers. This feature can be accessed using the clone() system call [7]. Unlike VM which runs a full operating system, a container can contain as little as a single process. If total isolation is not required, it is also easy to share some resources like storage among containers.

Docker achieves isolation of containers through the combination of four main concepts [4]:

Control groups

Namespaces

Stackable image layers

Virtual network bridge

I. Control groups (cgroups): This feature is used to manage the available hardware resources (memory, CPU, disk IO, network IO) and provides a convenient way for processes to access and utilize them. Cgroups can be used to create a subset of processes sharing a specific subset of resources thus providing hardware resource isolation.

The default linux kernel uses the Completely Fair Scheduler (CFS) which gives equal times to all the processes inside a single cgroup. This feature can be used in our tests to ensure that the bare metal processes and the Docker processes are treated fairly.

I*I. Namespaces:* Namespaces isolate and virtualize system resources like process IDs, user IDs, filesystems and network access. This makes the container OS unaware that it is running inside a virtualized environment. The process id's start with 1 inside the container and gets mounted with virtual root /.

III. Stackable image layers: Docker manages application installs and dependencies using image layers. Whenever a new application is installed in the container, an additional image layer would get added to the stack which would contain the new application files as well as the edited system files. These files are the changes to the previous layer.. This enables us to run a container completely independent of the things installed on the Docker host without wasting a lot of space.

IV. Virtual network bridge: This This component allows Docker to virtualize the network stack inside the container. Docker allows creation of a virtual network interface for each container which connects to the host network adapter. This network bridge can be used to expose specific ports to the outside world or even directly wire containers containers together through network without exposing any port to the outside world.

Now, we also need to consider how Docker tackles persistent storage the containers. There are two options that Docker allows: Bind Mounts and Volumes

I. Bind Mounts: When we use a bind mount, a file or directory on the host machine is mounted into a container. It has been available since that early days of Docker.

II. Volumes: This is similar to the way that bind mounts work, except that volumes are created and managed by Docker and are isolated from the core functionality of the host machine. Volume drivers can be used to store volumes to cloud drives or to encrypt the contents of volumes, or to add other functionality.

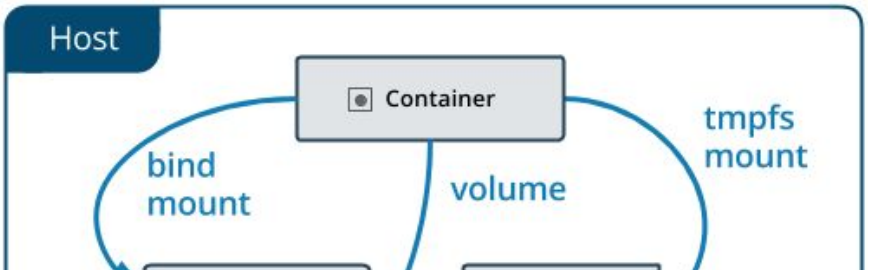

Fig 1: Different types of Docker storage showing volumes and bind mounts [6]

C. BENCHMARKS

I. STREAM: The Stream benchmark is a synthetic benchmark designed to measure sustainable memory bandwidth (in MB/s). This benchmark is specifically designed to work with data-sets much larger than the available cache on any given system, so that the results are more indicative of the performance of very large, vector style applications. [12]

Stream also has an MPI version which we use in this project since MPI is more relevant in the HPC space.

We have four tests in Stream, COPY, SCALE, ADD and TRIAD which are matrix manipulations as shown in the below figure [12].

Table I: STREAM Components

Name	Kernel	Bytes per iteration	FLOPS per iteration
COPY	$a[i] = b[i]$	16	0
SCALE	$a[i] = q * b[i]$	16	1
ADD	$a[i] = b[i] + c[i]$	24	1
TRIAD	$a[i] = b[i] + q * c[i]$	24	2

II. HACC IO: HACC IO is a synthetic benchmark for HPC systems which uses the IO kernel of the HACC (Hardware Accelerated Cosmological Code) simulation. The benchmark uses n-body simulation to simulate collision-less fluid in space. Each particle size is 38 bytes [9]. This benchmark mainly determines the storage performance (both reads and writes). HACC IO also uses MPI in its code.

III. Redis-Benchmark: Redis is a memory based key-value storage which is commonly used for caching, storing session information, and as a convenient way to maintain hot unstructured data sets. A Redis server puts a lot of pressure in the networking and memory subsystems. Since we run our benchmarks in a local environment, the main determinants would be the memory performance.

Redis includes the redis-benchmark utility that simulates running commands done by N clients at the same time sending M total queries (it is similar to the Apache's ab utility).

Redis server can be initialized with variable number of Io-threads.[13] This has been only possible since the latest version of Redis which is version 6.

There are various parameters that can be set while running the benchmark. They are as under:

Which command to benchmark : Redis has many commands both setters and getters and we can use both of those types in our benchmarks.

Number of concurrent clients: We can set the number of concurrent clients that can trigger Redis for the benchmark.

Number of times to trigger benchmark: This options allows us to set the number of times, a single command is triggered by a single client.

Pipeline size: Pipelining allows us to send multiple requests to the server without waiting for the replies.

Payload size: This allows us to set the amount of data to be sent per request in kilo bytes.

Number of threads: This allows us to set the number of threads that can concurrently trigger requests to the Redis server.

3. EVALUATION

IO overhead in Docker may either be because of memory or storage or network. In this project we try to find the memory and storage overheads in Docker environments comparing the performance to local applications. We investigate scenarios where one or more hardware resources are fully utilized and then we measure the workload metrics like bandwidth or requests per second to determine the overhead.

Our tests were performed on an Acer system with an Intel Core i7 9750H @ 2.60GHz processor with 6 cores and 16GB of RAM. The storage used is a Samsung NVMe SSD drive with read and write speed of around 3 GB/s. We used Docker version 19.03.13, Redis version 6.0.9 and MPICH as the MPI provider.
Additionally we used Ubuntu 20.4 LTS as the OS for both the local tests and for Docker containers to maintain consistency.

We didn’t create any additional cgroups for the containers and let them use the root cgroup so that they can use full system resources [5].

*I. Memory Bandwidth – STREAM:*In the four components of Stream, COPY, SCALE, ADD and TRIAD where once a page table entry is installed in the TLB, all data within the page is accessed before moving on to the next page. Hence the main determinants of performance are the bandwidth to main memory and to a much less extent, the cost of handling TLB misses.

For our tests we used the below combinations with Stream MPI:

Number of MPI processes: 1, 2, 4, 8

Number of array elements: 1M, 2M, 5M, 10M, 25M, 50M, 100M, 250M, 500M (M = 1000000)

The performance on Docker is almost identical to that of bare metal, with Docker volumes showing an overhead of 0.5 to 0.7% and Docker bind mounts showing an overhead of around 1 to 1.5%.

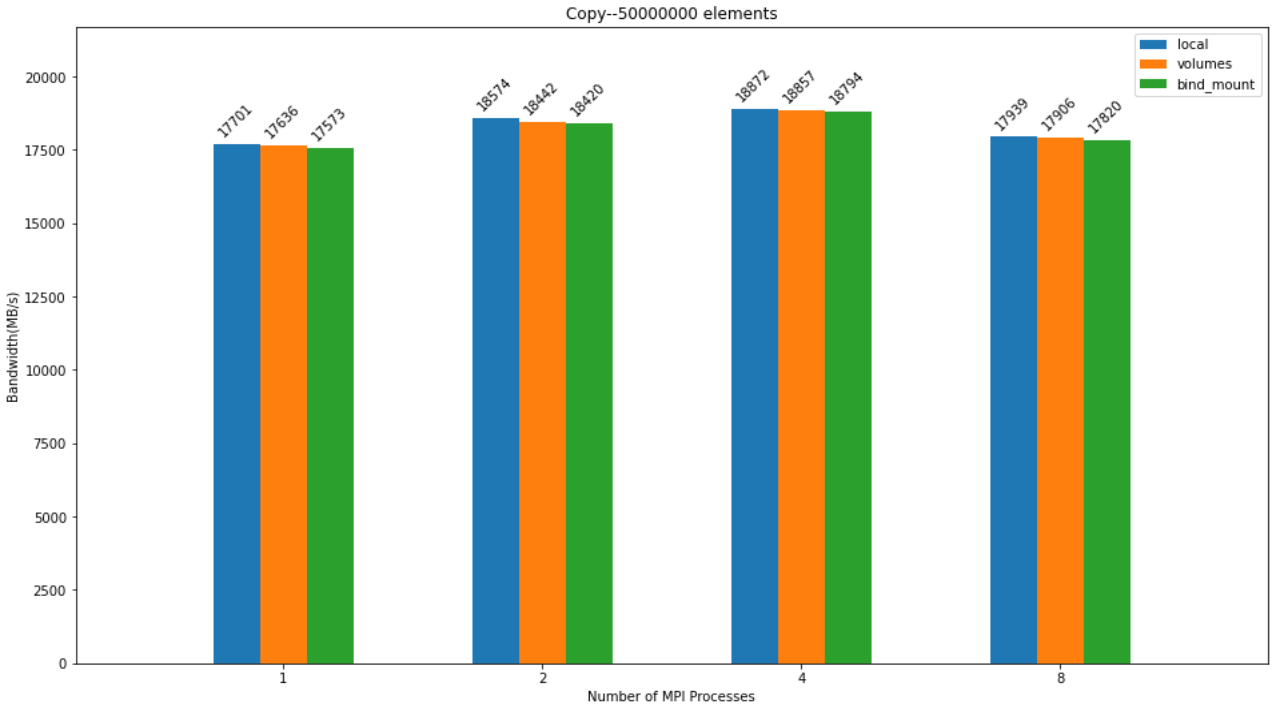

Fig 2: Comparing bare metal(local) with docker volumes and docker bind mounts for Stream COPY component with 50M elements equivalent to 762.5 MB

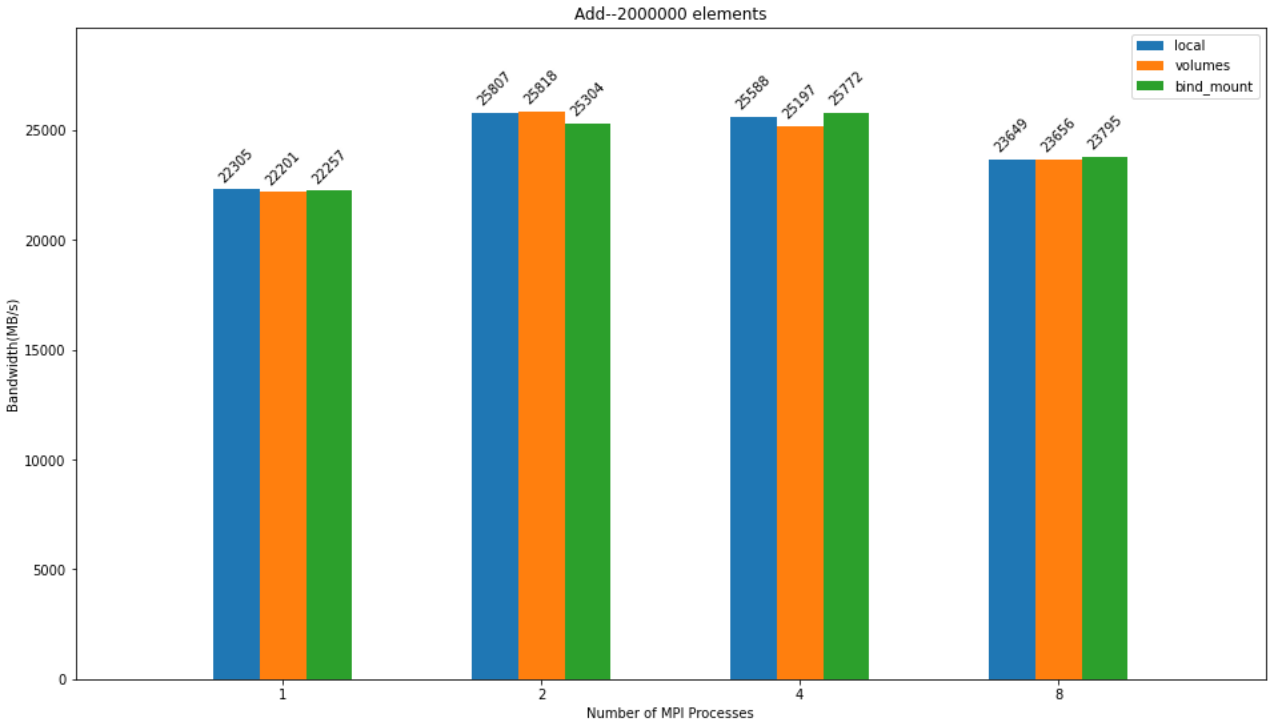

Fig 3: Comparing bare metal(local) with docker volumes and docker bind mounts for Stream ADD element of 2M elements equivalent to 45.7 MB

Fig 2 is a better indicator for memory bandwidth because the results are for a much bigger matrix size than the results in Fig 3.

Fig 3 should not be considered for memory bandwidth as it might be a better indicator for cache speed. Since the size of the matrix is small, it may be prone to high TLB hits. Hence the results are not consistent. Another explanation for the inconsistent results in Fig 3 may be due the OS scheduler for some reason has a bit more priority for containerized processes.

II. Block I/O – HACC IO: Block storage like SSD is commonly used in HPC systems for persistent storage. To test the overhead on such stems we use a Samsung NVMe SSD with read and write speeds of 3500MB/s and 3300MB/s respectively. The host file system is ext4. For Docker we use both bind mounts and volumes. Bind mounts uses the host filesystem and we mount the *~/Documents* directory to Docker. Volumes is Docker managed and the data gets stored in an object in */var/lib/docker/volumes/* which is the default location for linux.

For HACC IO benchmarks, we used the below combinations:

Number of MPI processes: 1, 2, 4, 8

No. of particles per rank: 1M, 2.5M, 5M, 10M, 25M, 50M (M=1000000)

To put things in perspective, 50M elements with 8 MPI processes mean a total IO read/write of 50*1000000*8*38 bytes = 14.15 GB. Each particle is first written and then read again from storage.

From the benchmarks, we see peak read speeds for single MPI processes and we see peak write speeds for 2 MPI processes. For reads, docker volumes have an overhead of 1 to 3.5 %. docker bind mounts have an overhead of 3% to 10% when compared to bare metal. For writes, the overhead for docker volumes is 1% and the same is 3.3% for docker bind mounts.

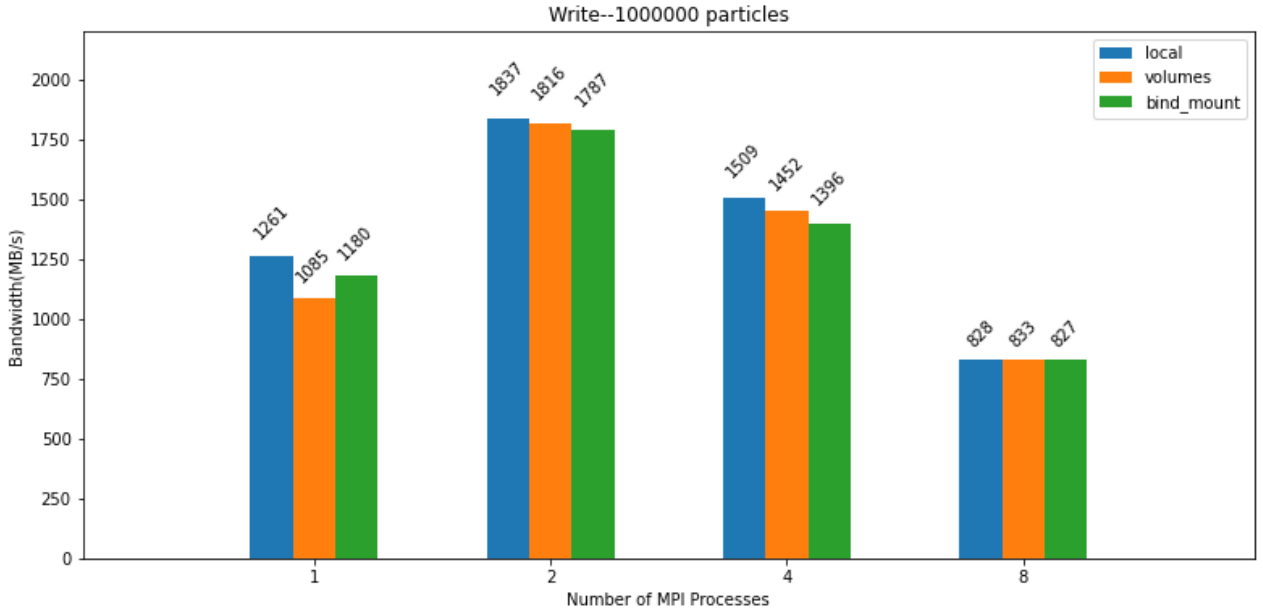

Fig 4: HACC IO benchmark for write of 1M particles equivalent to 36.25MB per MPI process

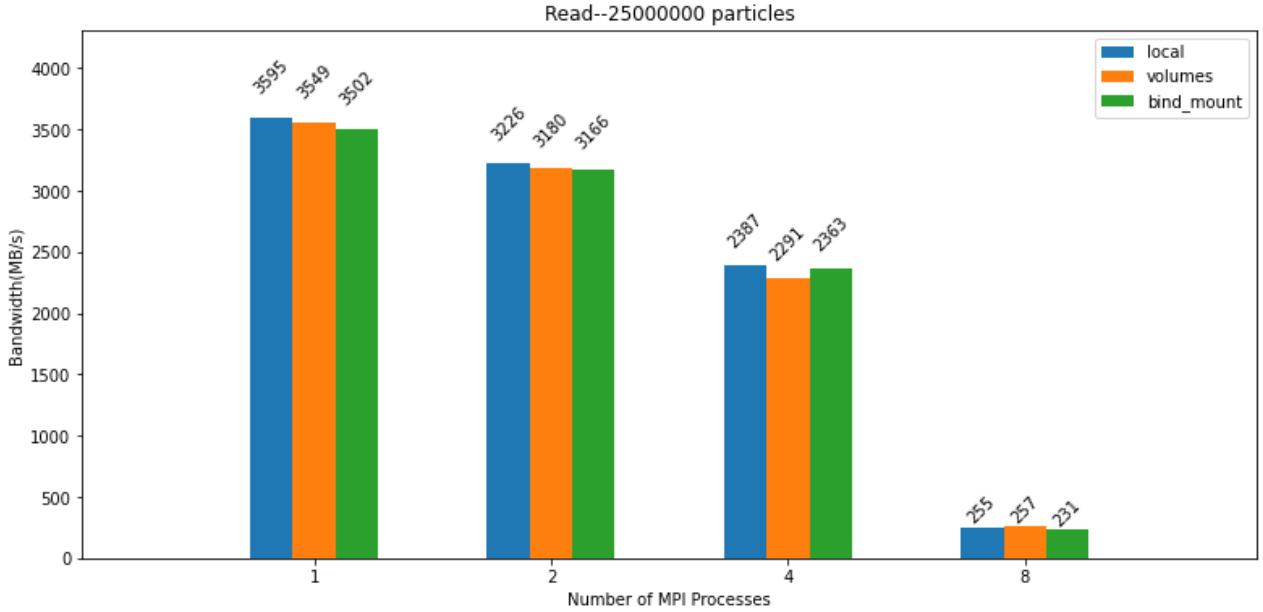

Fig 5: HACC IO benchmark for read of 25M particles equivalent to 906MB per MPI process

Fig 4 and Fig 5 shows that the storage IO overhead is very less for Docker containers. While the write behavior is similar to read behavior for bigger sizes, its is only for smaller sizes we see that two or more processes give better performance than a single process.

*III. Redis:*Redis being an in memory caching solution is used extensively in the HPC community. It has a collection of commands, some as simple as setting a value for a key and some others complex which take O(n) time.

The commands we benchmark are: GET, SET, MSET, HSET, LRANGE

SET and GET are O(1) actions. MSET is a O(n) action on a complex data structure. HSET is an action on a complex data structure which also takes O(1) time. LRANGE is for reading sequential data which takes O(S+N) time where S is the length of the list and N is the number of elements to retrieve.

All workloads were configured to produce/request objects of three different sizes 3 bytes, 100 bytes and 1000 bytes from the in memory Redis store. Additionally, pipelining of 1, 3 and 5 were used. The tests were done using 512 clients, with 1 million requests per client.

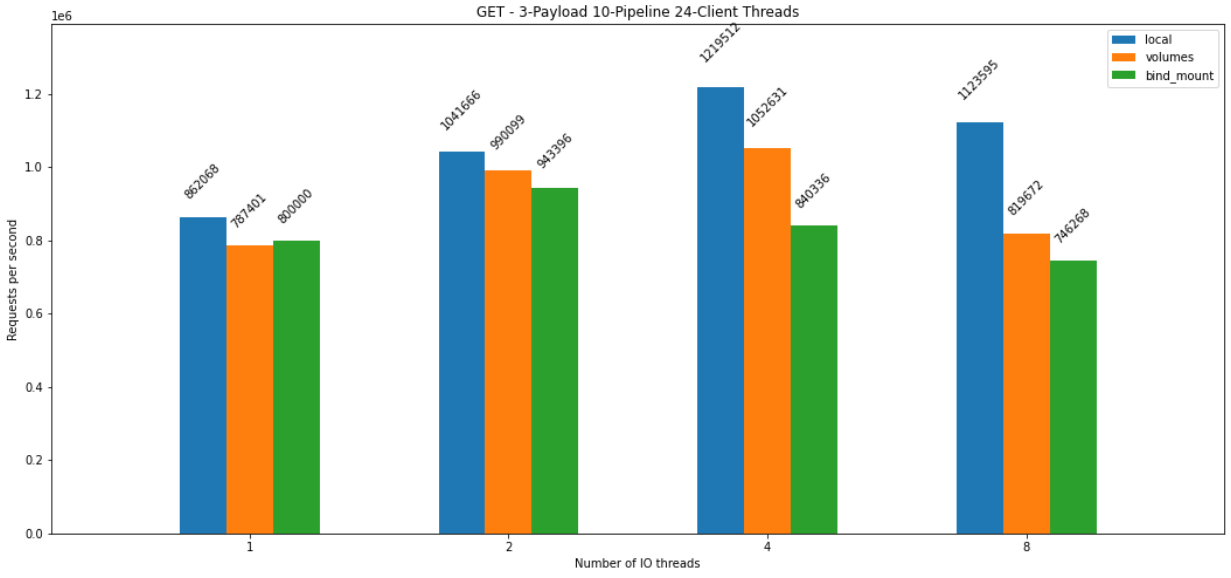

Fig 6: Redis GET with 3 byte payload and a pipeline of 10 requests and 24 client threads showing max number of requests per second of about 1.2M

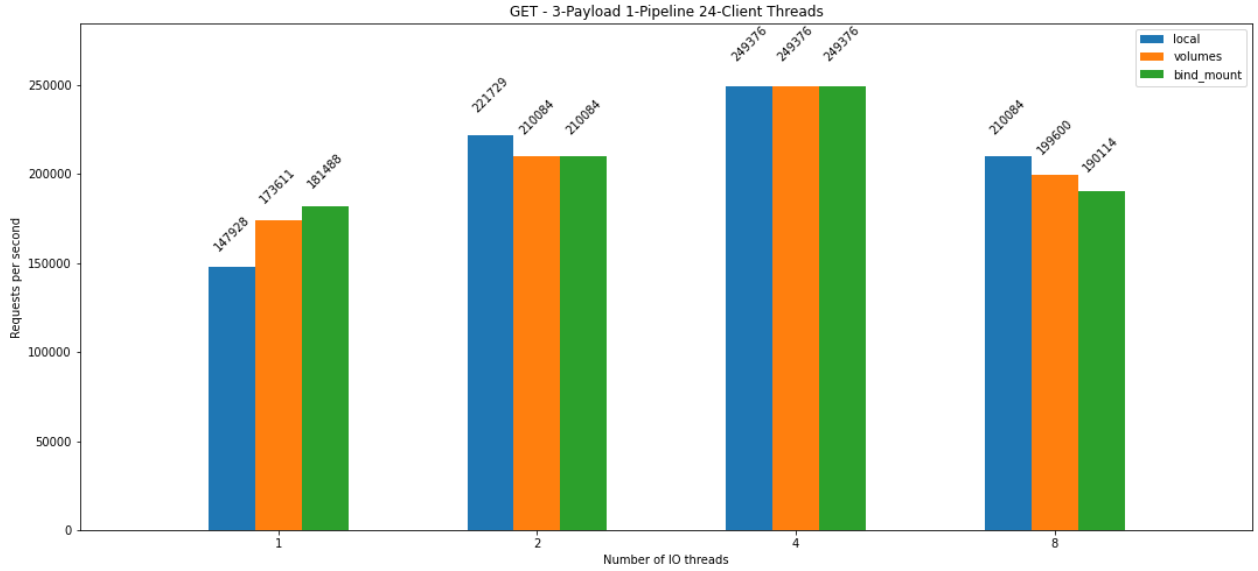

Fig 7: Redis GET with 3 byte payload without pipelining and 24 client threads showing max number of requests per second of about 0.2M

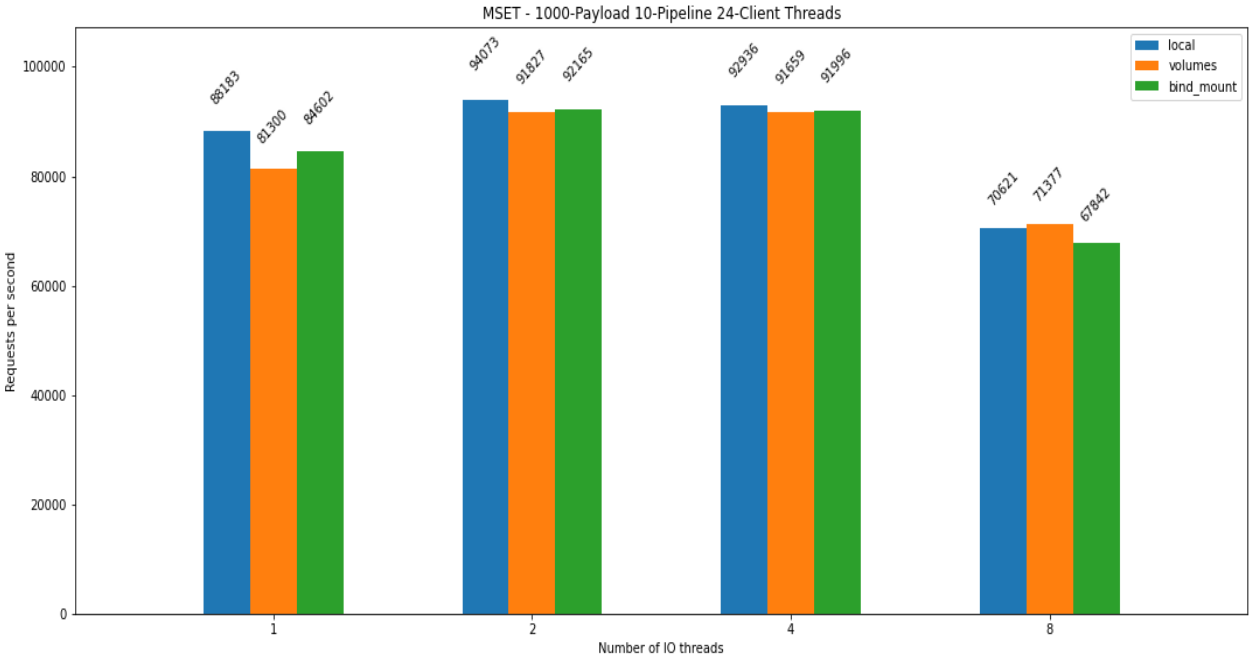

Fig 8: Redis MSET with 1000 byte payload and 10 pipeline requests with 24 client threads showing max number of requests per second of about 0.09M

Pipelining allows us to send multiple requests to the server without waiting for the replies. This leads to higher number of requests per second. Thus when we look at Fig 6 and Fig 7, we see that with pipelining of 10 requests the number of requests per second increased 5x in Fig 6 [13].

Using Fig 6 and Fig 7 we see that Docker volumes have an overhead of 5 to 27% and Docker bind mounts have an overhead of 7 to 33%. An overhead of 7% without pipelining leads to about 5* 7 ~ 35% overhead with pipelining.

Considering Fig 8, we see that as the payload increases and the data structure for the requests increases, the number of requests per second decreases and along with that the overhead decreases as well. The overhead in Fig 8 is around 2 to 4%. This stems from the fact that we are querying much less that the amount of requests shown in Fig 6 and Fig 7

4. CONCLUSION AND FUTURE WORK

Container technology has matured over the years and with the latest improvements in hardware and software we see that Docker has minimal memory and storage IO overhead when compared to a bare metal machine.

However, it is not to say that we can use Docker in all scenarios. As seen in the Fig 6 and Fig 7, a small amount of overhead for lower number of requests may lead to a very high overhead when the number of requests is multiplied.

These kind of problems can be ameliorated by using special techniques such as increasing the request payload and decreasing the number of requests as shown in Fig 8.

Our scripts and results are available online on github through the below links:

https://github.com/Ankur19/STREAM_Benchmark_Tests

https://github.com/Ankur19/HACC_IO_Benchmark_Tests

https://github.com/Ankur19//Redis_Benchmark_Tests

Since, we didn't see a high amount of overhead while considering memory and storage IO operations, it is perhaps the network IO operations that lead to the well documented high overheads.

Another important scenario to look at is having a swarm of docker containers run together and then running the benchmarks. This may lead to higher overheads because of the communication between the containers.

Additionally, with the introduction of rootless docker, we can hope that its use in the HPC community would increase and hence we should try to benchmark that version of Docker.

5. REFERENCES

[1] Container vs HPC – Mutually beneficial. [online] https://containerjournal.com/topics/container-management/containers-hpc-mutually-beneficial/

[2] Using Docker in High Performance Computing Applications - Minh Thanh Chung, Nguyen Quang-Hung, Manh-Thin Nguyen, Nam Thoai

[3] Container vs Virtualization. [online] https://www.brightcomputing.com/blog/containerization-vs.-virtualization-more-on-overhead

[4] Docker Demystified [online] https://blog.codecentric.de/en/2019/06/docker-demystified/

[5] Understanding Cgroups. [online] https://www.grant.pizza/blog/understanding-cgroups/

[6] Docker storage [online] https://docs.docker.com/storage/

[7] An Updated Performance Comparison of Virtual Machines and Linux Containers - Wes Felter et al - IBM Research

[8] Testing Docker Performance for HPC Applications - Alexey Ermakov, Alexey Vasyukov

[9] About HACC IO. [online] https://www.vi4io.org/tools/benchmarks/hacc_io

[10] Docker security. [online] https://docs.docker.com/engine/security/

[11] Docker rootless mode. [online] https://docs.docker.com/engine/security/rootless/

[12] About STREAM. [online] http://www.cs.virginia.edu/stream/ref.html

[13] Benchmarking Experimental Redis. [online] https://itnext.io/benchmarking-the-experimental-redis-multi-threaded-i-o-1bb28b69a314

[14] Performance Evaluation of Container-based Virtualization for High Performance Computing Environments - Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, Cesar A. F. De Rose

[15] HPC container runtime performance overhead: At first order, there is none - Alfred Torrez, Reid Priedhorsky, Timothy Randles

[16] Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using Intel Optane DC Persistent Memory Modules - Onkar Patil et al