Containerizing Parallel MPI-based HPC Applications

Antoine Schonewille
Saudi Aramco
Dhahran, Saudi Arabia
antoine.schonewille@aramco.com

Ahmed Bu-khamsin
Saudi Aramco
Dhahran, Saudi Arabia
ahmed.bukhamsin@aramco.com

***Abstract*—Software container technology based on Docker is a light weight packaging and virtualization technology. These containers are used to package and run the application with all its dependencies in a portable image with minimum requirements. Because of the complexity and dependencies of HPC (High performance computing) applications, this concept can be used to pre-package HPC applications. Although Docker is already used in cloud computing, HPC implementations are lagging due to, what we believe, the nature of most HPC applications: inter-process communications and the shortcomings of spawning containerized MPI-based applications on compute resources. This paper shows our solution for containerizing HPC MPI applications and running them across multiple hosts connected with Infiniband interconnect.**

I. Introduction

Docker containers is a new light weight approach to virtualization and application packaging and has the potential to be an HPC first class citizen tool. As shown by other works and confirmed by this paper, Docker containers deliver performance close to if not equal to bare-metal [2], [4]. Though Docker finds its purpose most commonly in Cloud computing, we broaden the scope of usability of Docker in the HPC world. In particular using Docker to package HPC applications and solve the issue of conflicting dependencies and simplifies dynamic HPC compute resources configuration. Large HPC systems usually serve large number of users who need to run different applications with conflicting requirements and dependencies. Instead of dividing the compute resources into partitions of different configurations to satisfy the different environmental requirements, the compute resources can be installed with minimal configuration and leave all environmental requirements to the application containers. This reduces the configuration management overhead and increases the overall compute resources utilization.

Nevertheless, as a new technology to the HPC domain, some HPC specific challenges need to be addressed, such as: the performance overhead for the different HPC workload types (compute bound, communication bound, memory bound, I/O bound), setting up and starting MPI workload for multi-host jobs, Configuring and utilizing Infiniband interconnect, integrating Docker jobs to the HPC work flow, scalability, security and user access control. In this paper, we are exploring how to run MPI workload using Infiniband on Docker containers without the need for adopting Docker aware cluster scheduler or resource manager. We are also benchmarking the MPI workload over Infiniband and Docker containers using High Performance Linpack [9] and Saudi Aramco's in-house developed reservoir simulator GigaPOWERS [3].

II. Related work

There have been several studies on HPC application performance and containers. Felter et al. executed multiple benchmarks to compare Docker containers performance to KVM virtual machine and native Linux. The results show that Docker containers introduce negligible overhead for CPU and memory performance while the network performance degrades when Docker NAT is used [4]. Gantikow et al. shows that running ABAQUS as a real world application on Docker containers on a single node using local disk and luster storage achieves close performance to the native execution on bare-metal with mean performance loss of 1.04% [7]. Higgins et al. investigated the performance of Docker containers by running Linpack on two nodes, using unspecified interconnect and found no considerable performance loss compared to native bare-metal execution. The researchers also proposed a method of executing MPI jobs in Docker containers by starting SSH daemon in every container to start the MPI process [8]. Ruiz et al. used NAS Parallel benchmark, using Ethernet interconnect, to study Docker containers performance using different Linux kernel versions, to reveal the performance of the Docker virtual network as interconnect [11]. Jacobsen et al. described how they deployed Docker into Cray XC cluster and the support for multi-node Docker jobs using custom SSH daemon [10]. Our paper differentiates from previous works in multiple aspects: we have benchmarked Docker containers using HPC standard HPL benchmark [9] running on multi-node using Infiniband interconnect, which is the most common setup in top500.com. We also run Saudi Aramco's reservoir simulator GigaPOWERS [3] as a real world application. In addition, we have proposed a method to launch Infiniband MPI-based applications without the need to start SSH daemon inside containers, which complies with security recommendations [5].

III. Background

A. Message Passing Interface

Message passing interface [6] is the most commonly used middleware in HPC for multi-node parallel jobs. MPI is a library of functions that can be called from different programming languages to provide various functionalities to parallel programs. Each MPI process stores its data in local memory and exchange data with other MPI processes, by passing messages through a network. There are many MPI implementations conform to the MPI standard. The standard defines different aspects of the message passing interface including the point-to-point message-passing, collective communications, and language bindings for C and Fortran.

There are multiple open-source MPI implementations such as Open-MPI and MVAPICH. There are also commercial MPI implementations available such as Intel MPI, ScaMPI (former HP-MPI) and Voltaire MPI. In this paper the evaluation experiments are conducted using MVAPICH2.

B. Docker Containers

Software container technology is a light weight packaging and virtualization technology [10] mostly applied in the cloud computing field. These containers are used to package the application with all its dependencies in a portable image, to make it easy to run on a platform or resource. It segregates the requirements for the Operating System from the requirements for the actual application. Third party software or application manufacturers can specify or build a certified container for their applications, reducing overhead for the customer. This segregation also allows for backward compatibility, where the container is running an older version of the Operating System as is the host. Docker container images can be generated through a Dockerfile which is a manifest or step-by-step description of tasks. An alternative is building an image interactively comparable to a "chroot"-environment. By default the started or initiated container is state-less. This ensures that every launch of the same instance (or image) is equal. Changed files or settings are not being maintained by default. Docker cleans up after an application ends by discarding the changes.

C. GigaPOWERS

GigaPOWERS [1] is a simulator that describes the oil, water, and gas flows inside the porous rocks that contain hydrocarbons. To do this, relevant physics, chemistry and thermodynamic relationships are integrated as highly complicated mathematical equations. The solutions to these equations are presented in grid blocks, or cells, that create a detailed visual picture of an oil field [2]. Cells can then be processes in parallel where inter-process communication is typically done through a low latency, high bandwidth interconnect like Infiniband.

[1]parallel oil, water and gas enhanced reservoir simulator
[2]http://www.oilandgasnewsworldwide.com/Article/33779/

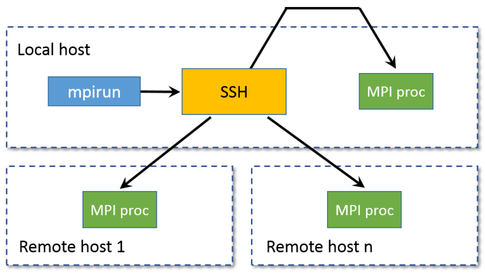

Fig. 1. Regular MPI application startup.

IV. Problem description

In HPC day-to-day operation, running MPI jobs on HPC clusters requires submitting a script that describes the job to the scheduler. The scheduler handles the compute resource reservation. With tightly integrated solutions, the scheduler handles the spawning of MPI processes. In other cases the job-script is responsible of spawning the required MPI processes. Either way requires MPI start up programs like "mpirun" or "mpiexec". These binaries are part of an MPI implementation. Most commonly, the host where the MPI start up program is being executed, which is the first host in the scheduler generated host-list, is responsible for initiating the remote MPI processes on the other hosts in the host-list through SSH connections. A graphical interpretation is shown in Figure 1.

After all MPI processes are spawned, the communications for the parallel application happens via the desired interconnect, such as Ethernet using TCP/IP or faster RDMA capable networks such as Infiniband. Although it is fairly straight forward to spawn and run MPI processes in a Docker container in a local host, scaling to other remote hosts is not a trivial problem. Equal to the method described earlier, the containerized MPI process started on the first host, which is responsible to spawn the remote MPI processes, needs to have a way to initiate SSH connections the remote containers. One solution [10] is to install the SSH daemon using a different port than the host and configure the container to use the host network and IP address, then SSH to the container using the host IP and the container specific SSH port directly. However, this solution raises multiple security concerns and increases the security management overhead [5]. While this could solve the SSH connection problem, the start-up of these remote containers is not solved by this approach.

The problem remains, how to spawn remote Docker containers along with the MPI processes inside these containers without running an SSH daemon in every container?

V. Proposed Solution

To avoid depending on third party solutions, like using a Docker aware scheduler to spawn remote containerized MPI processes, or having to start and run an SSH daemon in each Docker container, we came up with a simple yet elegant solution: an SSH client wrapper [1], made available in the container image. This wrapper, based on a small BASH-script replaces the SSH client binary in the container while the

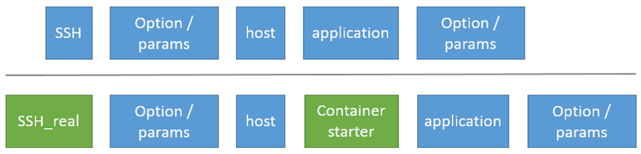

Fig. 2. SSH command-line before and after SSH wrapper.

original binary is being renamed and used at a later stage. The host Operating System was not modified.

A. SSH command-line reconstruction

The proposed SSH wrapper parses the SSH command-line that is being called by the MPI start-up program. This command-line typically comprises of five parts: the SSH client binary, SSH options and parameters, the remote host, the remote MPI application and the application options and parameters. As visualized in Figure 2, the wrapper splits up these parts and reconstructs the SSH command-line to call the renamed SSH binary, supply the SSH options and include the part to start the remote container. The remote instruction, the MPI application with its parameters, is supplied to this container-starter as a parameter.

The SSH wrapper also differentiates between local and outbound SSH connections, based on used remote host-name or IP-address. The local SSH connections requests do not require an actual SSH connection; the requested MPI application is spawned locally in the container. For the outbound SSH connections requests, the wrapper connects to the remote host using the reconstructed SSH command-line as seen in Figure 3. The SSH wrapper does not aggregate outbound SSH connections to the same host. For each outbound SSH session a remote container is being started. Recent MPI implementations do aggregation and as such spawn one outbond SSH session per remote host [3]. The container-starter, also a BASH-script, creates a new Docker container on the remote host and spawns the MPI application within that container. After the MPI processes stop or terminate, the spawned container is closed.

B. User environments and containers

For HPC applications to work, environments are typically configured in shell initialization files. This also applies for the containerized applications; the correct paths and variables have to be set in the container as well. One solution is to statically configure the environment in the Dockerfile. We used a different method and wrote an mpirun-wrapper called "docker-mpirun" [1] to prepare the environment and allow other run-time tuning like resource binding, described in Subsection VI-D. The mpirun-wrapper generates the "master_command" and "command" BASH-scripts. The master_command-script encapsulates the "mpirun" startup, while the command-script is an encapsulation of the MPI application. Both scripts are used to set the environment for the application at run-time: "master_command" inside the container on the first or local host; "command" inside the remote container. For optional run-time container tuning, the mpirun-wrapper also generates the container-starter script.

[3]Open-MPI and MVAPICH2 behave like described. Other MPI implementations might behave different and as such alterations might be required

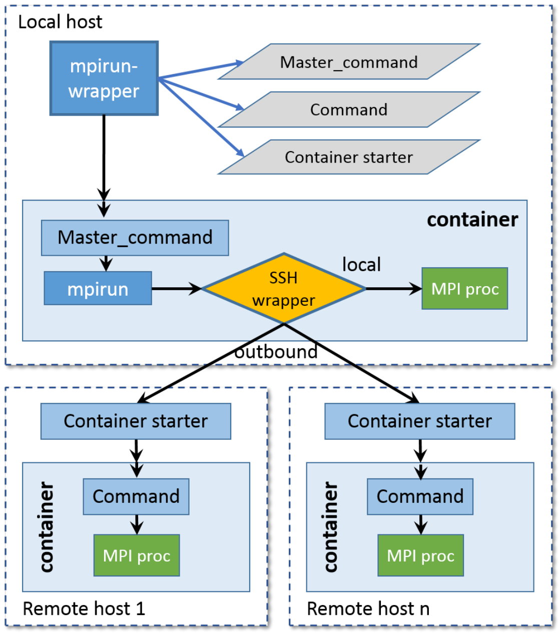

Fig. 3. Full Dockerized MPI application startup.

VI. EXPERIMENTAL EVALUATION

The primary goal of the experiment was to determine the effectiveness of the SSH wrapper approach in terms of scalability, usability and performance. We used a custom built MPI-based application to profile the container and MPI application start-up characteristics. The secondary goal was to measure the overall performance of a containerized solution. The performance was measured by means of conducting the well-known HPL or Linpack tests [9] and benchmarking GigaPOWERS [3], while using bare-metal [4] and containers.

A. Test Environment Setup

For the test, a cluster of 8 compute nodes was used. Each node was equipped with two Intel Xeon E5-2670 v2 10-core CPU, 64GB of memory and Qlogic QDR InfiniBand with full bisectional bandwidth as an RDMA interconnect.

The host Operating System for all nodes is RedHat Linux 7.2 64-bit Server Edition. The Docker images were built on RedHat 7.2 64-bit minimal image provided by RedHat. Different Docker images were built for different benchmark that include the SSH wrapper, MVAPICH2 and the benchmark specific files. Numactl and hwloc packages were included in both the host Operating System and the Docker image. Numactl allows running processes with a specific NUMA scheduling or memory placement policy [5] and is typically used to optimize or control the process and its memory location in multi-CPU architectures. The data for GigaPowers

[4]non-containerized
[5]numactl man pages

simulator were kept on GPFS filesystem mounted on the host and mapped to the container. When the container started it also mapped other files from the host such as the user home directory and the Infiniband device. The shared memory segment was set to 64 Gigabyte through the Docker –shm-size=64G [6] directive.

Both terms "process" and "rank" are being used. They are interchangeable, however we use rank to indicate just the MPI process, i.e. not a real application process.

Although the results presented here are obtained using MVAPICH2, Open-MPI was tried and proven to work as well.

B. *Startup Profile Results*

To measure the start-up time influence of Docker containers compared to bare-metal for a MPI-based parallel application, a small MPI test program was built to discover details regarding spawning the container and its MPI ranks. Both minimum of one and maximum of 20 ranks per node runs were profiled. The small MPI application was designed and benchmarked as depicted in Figure 4.

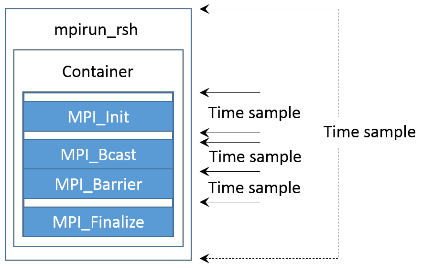

Fig. 4. Profiling Small MPI Application.

Overall timing measurements showed additional runtime for containers compared to bare-metal. The number of ranks or processes per node did not have much influence. Beyond two hosts, scaling-up ranks or hosts had even less effect on the additional consumed run-time, introduced by the start-up of the remote container. A near consistent 1.25 seconds were added to the total containerized runtime, compared to bare-metal. Figure 5 shows the runtime timing deltas in seconds for containers.

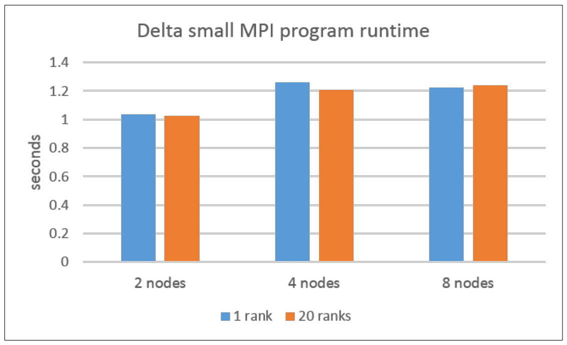

Fig. 5. Delta Containerized MPI Application Runtime.

[6]The total amount of available RAM available to the host

Containers compared to bare-metal showed little to no difference in MPI_Init timings. There was a timing difference measured for the 20-ranks-per-node benchmarks in regards to MPI_Barrier and MPI_Bcast visualized in Figure 6 and Figure 7 respectively. In this particular case it took longer for all ranks to do the MPI broadcast and to reach the MPI barrier, indicating a minimal delay in spawning all ranks in the remote containers. As the absolute numbers are very small compared to the total startup time, it had little to no influence. The benchmark did not show this delay for single-rank-per-node timings; the figures were near identical.

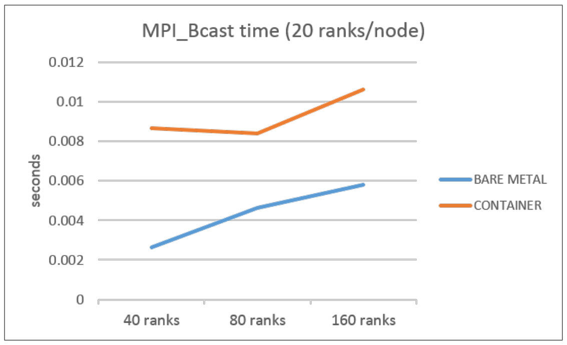

Fig. 6. MPI_Bcast timing for 20 ranks/node.

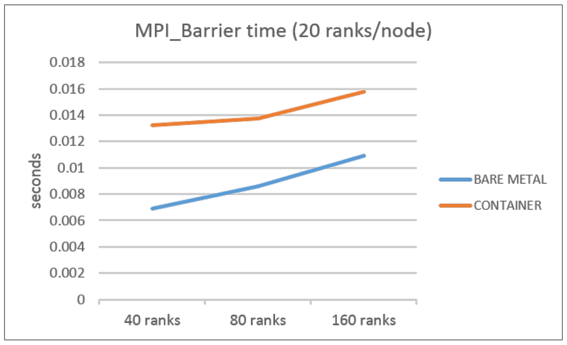

Fig. 7. MPI_Barrier timing for 20 ranks/node.

C. *HPL Benchmark Results*

TABLE I
HPL RESULTS

	Bare Metal	Containerized	Diff. %
1 host, 20 cores	350.3 GFlops	350.2 GFlops	-0.03%
2 hosts, 40 cores	702.9 GFlops	702.6 GFlops	-0.04%
4 hosts, 80 cores	1398.7 GFlops	1398.7 GFlops	-0.03%
8 hosts, 160 cores	2801.1 GFlops	2800.5 GFlops	-0.02%

The differences between bare-metal and the containerized approaches are minimal and negligible as visualized in table I.

D. *GigaPOWERS Benchmark Results*

GigaPOWERS showed better overall results when using 16 out of the available 20 cores per host due to assumed

Infiniband context limitations. This was the case for both containerized and bare-metal runs. As per default, Docker was able to see the host's 20 cores, however the MPI processes within the Docker container was set to 16. We maintained the 16 cores per host limit for all conducted GigaPOWERS benchmarks. The results in Table 2 were gained without any CPU core pinning or resource binding.

TABLE II
GIGAPOWERS RESULTS

	Bare Metal	Containerized	Diff. %
1 host, 16 cores	7.683 hours	7.689 hours	-0.08%
2 hosts, 32 cores	4.070 hours	4.118 hours	-1.18%
4 hosts, 64 cores	2.223 hours	2.245 hours	-0.99%
8 hosts, 128 cores	1.216 hours	1.227 hours	-0.90%
16 hosts, 256 cores	0.740 hours	0.748 hours	-1.07%

Although containerized GigaPOWERS showed linear scaling characteristics, the runs are slightly slower (<1.2%) as compared to bare-metal, with the exception for a single-host run (16 cores). In this specific case containerized and bare metal runs show similar timing results.

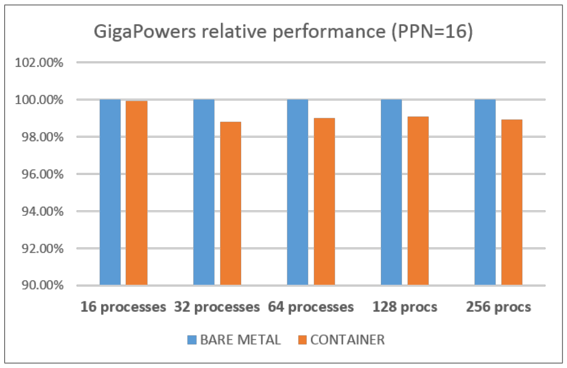

Fig. 8. GigaPOWERS Relative Performance Results.

To better explain the slight increase in runtime, high level function-call timings were applied. This revealed the additional time was spent on Memory synchronisation: MPI_Alltoall and MPI_Allgather calls. In parallel to this, we observed that specific MVAPICH2 runtime or command-line parameters were ignored or did not have any effect inside a container. Specifically the following options that set process affinity or map processes to CPU-s or cores: MV2_ENABLE_AFFINITY [7], MV2_CPU_BINDING_LEVEL and MV2_CPU_BINDING_POLICY. It was assumed, yet this needs to be confirmed, that Docker overruled these directives while having more knowledge of the underlying hardware or resource. As an alternative to MPI native affinity functionality and to confirm the influence of affinity, CPU core pinning or resource binding, additional functionality was added to our SSH wrapper. With the containerized runs, two techniques were tried:

[7]enabled by default as per MVAPICH2 documentation

- Using numactl within Docker instance, applied to the MPI processes itself.
- Native CPU core pining of Docker instance using the –cpuset-cpus directive. Processes inside the docker instance were left as-is.

CPU core pinning of MPI processes with numactl inside the Docker container had both positive and negative impacts, depending on the activity and core location (CPU 1 vs. 2). I/O and interconnect performance improved compared to non-pinned or CPU core balanced runs. This was true when the CPU with more used cores shared the same PCI-bus as the Infiniband HCA [8] or Ethernet adapter. On the other hand CPU intensive tasks or function calls showed a degradation. In our case the Docker native core pinning approach only had negative influences and was dropped in further benchmark runs. Since all available cores were used during the HPL benchmark runs, no affinity or pinning was tried there.

Using the Docker –shm-size=64G directive made a big difference in performance. The default of Docker is 32GB which has some indications of preferring memory of one processor, while a more balanced utilization was expected. This then affects the performance of shared memory tasks due to NUMA characteristics.

Similar to the HPL benchmarks, GigaPOWERS showed a linear scaling pattern for both bare-metal and containerized benchmarks.

VII. Conclusion

We successfully implemented a third-party independent solution that allows containerizing HPC/MPI-based parallel applications and running these across multiple hosts while utilizing the host available Infiniband interconnect network. This has been achieved with very comparable performance to regular bare-metal approaches.

The benchmarked applications scale near equally well for Docker containerized approaches as compared to regular bare-metal environments. Our solution showed only non-influential or marginal overhead of start-up timings compared to bare-metal.

VIII. Future work

The Docker containerized approach has more potential that need to be looked into. One of these potentials is live-migration [4] or moving a running Docker instance to another host. This functionality can be of great benefit to the HPC community as it allows for migrating off of a possible failing compute resources to another available resource. The benefits are that larger or longer running parallel applications can be kept running without being interrupted or terminated. This enhances the reliability of parallel applications and HPC resources or clusters in general.

Other pending activities include further application optimization, including specific resource or domain binding (core pinning), to further enhance the application performance.

[8]Host Channel Adapter: Infiniband adapter

Core pinning or domain separation will also be required in environments where multiple jobs share the same resources to allow for more fair resource sharing.

Large scale runs: To ensure scalability, larger runs need to be performed while using the discussed approach.

References

[1] A. Bu-khamsin and A. Schonewille. Docker for hpc. *https://github.com/ambu50/docker-ib-mpi*, April 2016.

[2] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame. The impact of docker containers on the performance of genomic pipelines. *PeerJ*, 3:e1273, 2015.

[3] A. H. Dogru, L. S. Fung, T. M. Al-Shaalan, U. Middya, J. A. Pita, et al. From mega cell to giga cell reservoir simulation. In *SPE Annual Technical Conference and Exhibition*. Society of Petroleum Engineers, 2008.

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[5] C. for Internet Security. Cis docker 1.11.0 benchmark. *https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.11.0_Benchmark_v1.0.0.pdf*, 2016.

[6] M. P. I. Forum. Message passing interface. *https://www.mpi-forum.org*.

[7] K. S. Gantikow H. and R. C. Container-based virtualization for hpc. *In Proceedings of the 5th International Conference on Cloud Computing and Services Science*, pages 543–550, 2015.

[8] J. Higgins, V. Holmes, and C. Venters. Orchestrating docker containers in the hpc environment. In *International Conference on High Performance Computing*, pages 506–513. Springer, 2015.

[9] HPL. High performance linpack. *http://www.netlib.org/benchmark/hpl*.

[10] D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 2015.

[11] C. Ruiz, E. Jeanvoine, and L. Nussbaum. Performance evaluation of containers for hpc. In *European Conference on Parallel Processing*, pages 813–824. Springer, 2015.