2017-06-29

Distributed Checkpointing with Docker Containers in High Performance Computing

Gustaf Berg **Magnus Brattlöf**

DEGREE PROJECT
Computer Engineering
Bachelor level G2E, 15 hec

Department of Engineering Science, University West, Sweden

Distributed Checkpointing with Docker Containers in High Performance Computing

Sammanfattning

Container-virtualisering har blivit mer och mer använt efter att uppdateringar till cgroups och namespace-funktionerna släpptes i Linuxkärnan. Samtidigt så lider industrins högpresterande beräkningskluster av dyra licenskostnader som skulle kunna hanteras av virtualisering. I den här uppsatsen utformades experiment för att ta reda på om Dockers funktion checkpoint, som fortfarande är under utveckling, skulle kunna utnyttjas i industrins beräkningskluster. Genom att demonstrera detta koncept och dess möjligheter att pausa distribuerade containrar, som kör parallella processer inuti, användes den välkända NAS Parallel Benchmarken (NPB) fördelad över två test-maskiner. Sedan så pausades containrar i olika ordningar och Docker lyckas återuppta benchmarken utan problem både lokalt och distribuerat. Om man försiktigt överväger ordningen som man skriver ner containers till disk (checkpoint) så går det utan problem att återuppta benchmarken lokalt på samma maskin. Slutligen så visar vi även att distribuerade containrar kan återupptas på en annan maskin än där den startade med hög framgång. Dockers prestanda, möjligheter och flexibilitet lämpar sig i framtidens industriella högpresterande kluster där man mycket väl kan köra sina applikationer i containrar istället för att köra dom på det traditionella sättet, direkt på hårdvaran. Genom användning av Docker-containers kan man hantera problemet med dyra licenskostnader och prioriteringar.

Datum:	2017-06-29
Författare:	Gustaf Berg Magnus Brattlöf
Examinator:	Thomas Lundqvist
Handledare:	Andreas de Blanche
Program:	Nätverkstekniskt Påbyggnadsår, 60 hp
Huvudområde:	Datateknik
Utbildningsnivå:	Grundnivå
Kurskod:	EXD500, 15 hp
Nyckelord:	*Industrial HPC, HPCC, Suspend, Pause, Checkpoint, Docker, CRIU.*
Utgivare:	Högskolan Väst, Institutionen för ingenjörsvetenskap 461 86 Trollhättan Tel: 0520-22 30 00 Fax: 0520-22 32 99, www.hv.se

Distributed Checkpointing with Docker Containers in High Performance Computing

Gustaf Berg, Magnus Brattlöf
Bachelor thesis in Computer Engineering
University West
Trollhättan, Sweden
{firstname.lastname}@student.hv.se

***Abstract*—Lightweight container virtualization has gained widespread adoption in recent years after updates to namespace and cgroups features in the Linux kernel. At the same time the Industrial High Performance community suffers from expensive licensing costs that could be managed with virtualization. To demonstrate that Docker could be used for suspending distributed containers with parallel processes, experiments were designed to find out if the experimental checkpoint feature is ready for this community. We run the well-known NAS Parallel Benchmark (NPB) inside containers spread over two systems under test to prove this concept. Then, pausing containers and unpausing them in different sequence orders we were able resume the benchmark. After that, we further demonstrate that if you carefully consider the order in which you Checkpoint/Restore containers, then the checkpoint feature is also able to resume the benchmark successfully. Finally, the concept of restoring distributed containers, running the benchmark, on a different system from where it started was proven to be working with a high success rate. Our tests demonstrate the performance, possibilities and flexibilities of Dockers future in the industrial HPC community. This might very well tip the community over to running their simulations and virtual engineering-applications inside containers instead of running them on native hardware.**

***Index Terms*—Industrial HPC, HPCC, Suspend, Pause, Checkpoint, Docker, CRIU.**

I. Introduction

Expensive licensing costs, energy use, and limited hardware resources often lead to systems where users must share resources. Proprietary software, like industrial simulations or virtual engineering can add significant cost to a company in terms of licensing. These licenses are usually tied to hardware like the number of cores in High Performance Computing (HPC) systems.

While it is basically standard practice to use virtualization with cloud providers and datacenters around the globe, the industrial HPC-community have not yet migrated from running applications on native hardware. This is mainly because of the overhead of full Virtual Machines (VMs) that run on top of hypervisors. For example, each VM has its own kernel that comes with a (guest) operating system. In turn the hypervisor has its own kernel too. This effectively makes applications installed on the VM communicate through two kernels, adding unnecessary overhead and degrading application performance.

In comparison to VMs, container virtualization is a lightweight alternative, which gained traction among consumers after updates to control groups (cgroups) and namespace-features in the Linux kernel. Containers also provide file system, network and process isolation as VMs do. Resource control of memory, disk and CPU are done through cgroups without the communication-overhead of an additional kernel. This allows for effective and dense deployment of services with near-native performance and most importantly the many benefits of virtualization can be leveraged with low overhead. The popular container management software Docker [1] has a built-in pause and a experimental Checkpoint/Restore (C/R) functionality that the HPC-community could benefit from and this needs to be investigated.

One problem that could occur in an industrial HPC cluster is that employees working during office hours (09:00-17:00) have to share expensive hardware resources and licenses. The problem manifest itself when these shared resources are not readily available, (i.e., a new higher-priority industrial simulation needs to be scheduled). When this happens there are some options that could solve this problem. The first would be waiting for the lower prioritized simulation to finish, but that could cost the company lots of money. Another option would be to kill the lower-priority simulation. However, that would force the simulation to be restarted all over again and could potentially mean wasting billions of CPU-cycles, time, energy and resources. Another way to alleviate these problems is scheduling many short and long-running jobs during off-hours (18:00-06:00), but this is not a complete fix. The same problem could still occur, unfinished jobs that are not complete by morning (06:00) still risk being killed. Virtualization allows for VMs and containers to be to suspended. The long-running and many short jobs that are not finished by morning could be suspended during office-hours, allowing them to be restored again whenever there are free resources available during off-hours without simulations and applications having to be restarted from the beginning.

There are several mature and robust Checkpoint/Restart software available on the market to checkpoint parallel processes running on native hardware [2]. The most widely ones used are Distributed Multi-Threaded Checkpointing (DMTCP) [3], Berkeley Lab Checkpoint/Restart (BLCR) [4] and Checkpoint/Restore In Userspace (CRIU) [5].

This thesis will focus on the jobs that risk being killed and will contribute with an evaluation of the ability to pause and resume containers by testing the experimental Checkpoint/Re-

store feature of Docker (17.03.1-ce). Checkpointing containers could potentially tip, at least, the industrial HPC-community over to running some or all their simulations in containers in the future instead of running them on native hardware. Making more effective use of available resources while still managing expensive software licensing costs. To the best of our knowledge, there have not been any studies exploring the possibility of checkpointing distributed containers with parallel processes inside Docker containers.

The rest of the thesis is structured as follows. We explain our Experimental Setup in Section III, then follows Design and Baseline in Section IV. Pausing and Resuming containers is found under Section V. Checkpoint and Restore is located in Section VI. Checkpoint, Move and Restore is found in Section VII, then Checkpoint, Switch and Restore can be found under Section VIII. Finally, we finish with Discussion and Conclusions in Section IX and Section X respectively.

II. Related Work and Containers

Containers are becoming a powerful competitor against hypervisor-based virtualization in many areas. Hypervisor-based virtualization such as Kernel Virtual Machine (KVM) [6] and VMware [7] virtualizes hardware and device drivers which makes it possible for multiple guest operating systems to run on a single host. There has been many previous studies about hypervisor-based virtualization and containers [8], [9], [10] and how they differ in architecture and performance. In [8] W. Felter et.al. explored the performance of traditional VMs and compared it with the use of Linux containers. They reported that containers equals or exceeds VMs in every test case that they conducted.

Linux containers use the concept of operating-system-level virtualization, which is a technique built on top of the namespaces [8] and control groups (cgroups) [11] functionalities of the Linux kernel. The isolation between containers are maintained by creating hostname, filesystem, network, Process ID (PID), user, and InterProcess Communication (IPC) namespaces. Processes running within a container appears to be running on a regular Linux system with their own resources, however they are all sharing the same kernel with other processes located in other namespaces. By grouping processes together, cgroups allows for managing aggregated resources, (i.e., constraining CPUs and memory for a specific container). Containers can see available resources from the host system but they are not aware of the resource limits they might have [8].

In [9] Morabito et.al. focused on strengths, weaknesses and anomalies in traditional hypervisor-based virtualization compared to the more lightweight alternative, containers. Running benchmarks that were I/O, memory, network and CPU intensive, they concluded that the level of overhead introduced by containers could be considered almost negligible and that containers allows for a more dense deployment than traditional VMs. Figure 1 illustrates their differences in architecture.

M.G. Xavier et.al. [12] performed an in-depth performance evaluation of different container virtualization techniques.

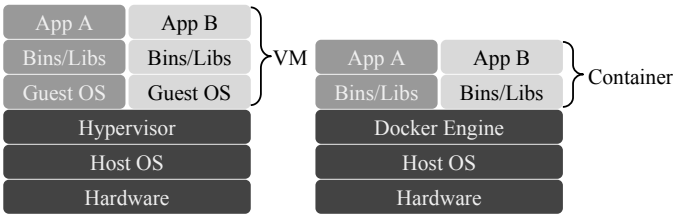

Figure 1. Hypervisor-based virtualization is illustrated to the left and to the right container-based virtualization is shown, and how they differ in architecture.

They argued for the architecture behind containers could be a powerful technology for HPC-environments. They also argued that HPC-environments can only take advantage of hypervisor-based virtualization if the fundamental CPU, memory, disk and network overhead is reduced. Their research demonstrated that all container-based techniques tested had near-native performance of system resources. Finally, their conclusion was that there is a lack of flexibility to containers regarding live migration, checkpoint, and resume functions in the kernel.

In [13] J. Higgins et.al. describe how to implement a container orchestration model for HPC-clusters. Either all processes are scheduled within one container (Model 1) on nodes, or all processes are scheduled with one container per process (Model 2) on nodes, which is illustrated in Figure 2. They also claim that "there is no difference in performance between the two container orchestration models proposed."

Docker [1] is an open-source software container platform written in the programming language Go [14] and originated from many previous container-technologies explained in C. Boettinger [15], such as Linux Containers (LXC) [16]. In [8], W. Felter et.al. describes that Docker has rapidly become a well-known and standard management tool for Linux containers due to its rich feature set and ease of use. Management and configuration of containers are handled by *Docker engine*, which is built in three-layers. A daemon, a CLI and a REST API. The latter is used for communication between the daemon and CLI. Docker containers are an executable instance of a pre-built image, which in turn is a piece of software that includes everything that is needed to run on a host machine (e.g., system tools, libraries and settings). The image is built using a compressed Linux distribution (e.g., CentOS, Ubuntu or Fedora).

Communication in parallel computing usually happens with the tried and true standard protocol Message Passing Interface (MPI) [15]. This protocol was developed for sending and receiving messages between tasks or processes. MPI synchronizes tasks and processes with a process-launcher and can perform operations on data in transit. Freely available implementations of MPI include OpenMPI [17] and MPICH3 [18].

The traditional way of suspending processes running on a Linux system is to use the SIGSTOP and SIGCONT signals. This is not always sufficient for userspace because it is noticeable by parent processes in a *waiting* or *ptrace* state. The signal SIGSTOP cannot be caught by the task itself

while SIGCONT can [19]. This can lead to problems where these signals are broken by other programs trying to stop and resume tasks. There is however, a built in command `docker pause` that suspends a running container, and its processes. On Linux this command leverage cgroup freezer [11] that use the kernel freezer code to prevent the freeze/unfreeze cycle from being visible to tasks being frozen. All affected processes are unaware that they are in a frozen state and the state is not visible by the tasks being frozen [20]. Working hierarchically, freezing the cgroup, also freezes descendants. Each cgroup has its own parent-state and a self-state. If both of these states are thawed then the cgroup is also thawed.

The checkpoint function in Docker is an experimental feature requiring Linux kernel 3.11 or higher. This feature is based on Checkpoint/Restore In Userspace (CRIU) [5]. CRIU provides the ability to freeze running applications and their process trees, writing them to persistent storage as a collection of one or more image files. These image files contain memory pages, file descriptors, inter-process communication and process information from the /proc file system [21]. First a process dumper collects threads walking through /proc/$pid/task directory and /proc/$pid/task/$tid/children gathering recursive information from the child processes. Reading all the information that it knows about these collected tasks, CRIU then dumps them to image files that can later be restored.

III. Experimental Setup

Experiments were conducted to investigate the possibility of running parallel processes inside distributed containers that communicate with MPI, as a proof of concept targeting shared resources and expensive licensing costs found in the industrial HPC-community. By running parallel processes inside distributed containers would allow for jobs to be suspended and restored later. To that end, we designed a series of five container experiments that start with a demonstration of container-performance compared to running applications on native hardware. The five experiments are detailed in Table I.

All our tests and experiments were performed on two identical and independent testbeds (Alpha and Beta), each equipped with a 3.2 GHz Intel i5-3470 Ivy Bridge processor with a total of four cores per processor and 8 GB of DDR3 RAM. The HDDs were two Seagate Desktop 500 GB, with 16 MB Cache and connected with SATA 6.0 Gb/s. The CPU-power governor was configured in performance mode together with min-frequency-scaling set to the same value as maximum to prevent frequency scaling. Intel turbo boost technology was also turned off to ensure equal and consistent frequencies throughout tests. The testbeds ran CentOS 7.3 with kernel release 4.10.2-1.

The well-known suite from NASA, Numerical Aerospace Simulation (NAS) Parallel Benchmarks (NPB) [22], "derived from Computational Fluid Dynamics (CFD) applications, mimic computation and data movement", was compiled to be used with MPICH3. The benchmark class C of NPB 3.3, standard test problems, was used together with Conjugate Gradient (CG); irregular memory access and communication to put the systems under workload long enough to be able pause and checkpoint containers.

TABLE I
THIS TABLE OUTLINES FIVE EXPERIMENTS THAT WILL BE CARRIED OUT TO DEMONSTRATE THE SUSPENSION OF CONTAINERS RUNNING PARALLEL PROCESSES THAT COMMUNICATE VIA MPI.

Section	Container Experiment	Description
IV.	**Design and Baseline**	*Include necessary dependencies in the container and build. Run two, four and eight (4+4) parallel processes and compare execution time with with native baseline.*
V.	**Pause and Unpause**	*Docker pause two, four and eight (4+4) running parallel processes inside containers. Docker unpause and verify successful completion of the benchmark.*
VI.	**Checkpoint and Restore**	*Using the built in experimental feature docker checkpoint, suspend two, four and eight (4+4) parallel processes. Restore the state and finish the benchmark successfully.*
VII.	**Checkpoint, Move and Restore**	*Checkpoint the running, parallel HPC application, with two, four and eight (4+4) processes. Move the image files to a different testbed and restore them there. Verify successful completion.*
VIII.	**Checkpoint, Switch and Restore**	*Checkpoint the HPC application with two, four and eight (4+4) parallel processes inside. Restore images on the opposite testbed and verify successful completion.*

IV. Design and Baseline

The base container image of CentOS 7 (`centos:latest`) from Dockerhub [23] was used as a base to build an image. Then, a Dockerfile, listed in Appendix A, was designed and created to automate the building of a complete image to run NPB CG.C. The image includes all necessary libraries, binaries and dependencies to run MPICH3.

Since MPI use SSH to communicate with other processes and tasks, the path for configuration of SSH were included in the Dockerfile. The command `docker build -t <mpi_image> </path/to/Dockerfile>`, created the image where -t flag tags the image with a suitable name. Multiple working containers could then be started from the same image.

Configuring and compiling HPC-applications inside containers to use MPICH3 could also be automated with the Dockerfile, but it was not needed. All containers running on their respective testbed shared a Docker volume `/hpc` where all software was located, pre-compiled and configured for the individual testbed. This volume was automatically mounted at startup in all containers.

When processes or tasks inside containers communicate via MPI it can use hostname-to-IP lookups. If the corresponding hostname (in the container) was not present in /etc/hosts, communication could be unsuccessful and execution would then be interrupted. This problem occured when containers were restarted or if the host system was rebooted, which led to configuration in /etc/hosts being cleared. A fix for this minor problem was to manually add the additional parameter `--add-hosts=<hostname>:<IP>` when deploying containers. This allowed for address consistency and persistence in /etc/hosts for each container, even if the host system was rebooted or containers were restarted. Automatically adding these parameters, when launching a job and creating

containers on-demand, could be integrated to a job scheduler, like SLURM [24].

To run distributed Docker containers over the two testbeds, Alpha and Beta, static IP routes were reconfigured manually to direct communication between testbeds and containers. All containers were pre-deployed in our experiments, however starting and stopping containers on-demand could also be implemented to the job scheduler of choice.

A. Container Orchestration Models and Declaration

Two orchestration models will be tested in accordance to J. Higgins et.al. [13]. The first orchestration Model 1, placed all running processes within one container per testbed, (i.e., four processes within one container). The second Model 2, placed all processes with a one-to-one mapping of containers (i.e., four processes with four containers or eight processes with eight containers). The two orchestration models are presented in Figure 2 below.

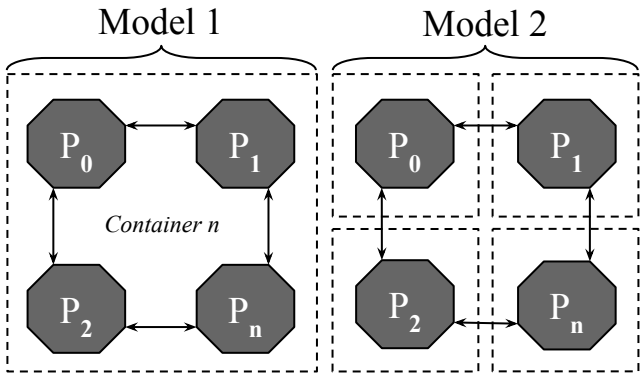

Figure 2. Two orchestration models for scheduling parallel processes from J. Higgins et.al. [13]. Either schedule Processes (P) according to Model 1, which run several processes inside each container. Or schedule them according to Model 2, which use one container per process. Arrows indicate process communication in containers for Model 1 and between containers for Model 2.

Orchestration Model 1:

$$Containers_n = Nodes_n \tag{1}$$

Orchestration Model 2:

$$Containers_n = Processes_n \tag{2}$$

All container experiments will be declared and follow the same approach throughout the rest of the thesis, if nothing else is stated. Two (CG.C.2) and four (CG.C.4) parallel processes will be scheduled to execute in either one container according to Model 1 or in one container per process according to Model 2, each done sequentially, one at a time on testbed Alpha. Eight parallel processes (CG.C.8) will be spread evenly over the two testbeds Alpha and Beta, in two containers (1+1) with Model 1. Consistency of tests were ensured with the help of Python scripts, looping through each tests. Scripts that were used are listed in Appendices B, C and D.

B. Baseline Creation

The NPB CG benchmark with two, four and eight processes was executed on native hardware. NPB automatically does a verification calculation after each run, and reports the execution time a specific benchmark and class took to finish. This time was used to create a baseline on native hardware and used to compare running the same benchmark inside Docker containers. The baseline was used to control conclusions made by M. G. Xavier et.al. in [12] and conclusions made by R. Morabito et.al. in [9], that containers do not add unnecessary overhead and approximately perform at native speed when it comes to high workloads.

C. Baseline Results and Analysis

Figure 3 presents the results from running Conjugate Gradient class C with the average native execution times normalized to one, in comparison to using Docker containers. For two processes, Model 1 performed better than Model 2, but all runs had a high standard deviation (4,6) so results fluctuated a lot which is illustrated in the figure with error bars. Comparing Native execution times with four processes, a similar pattern was found and the standard deviation (0,3) was much lower. However, when running eight processes and Model 1 the benchmark was 30% slower than Native, with a standard deviation of 7,7. Model 2 had the lowest standard deviation of every test (0,038) and was only 0,5% slower than executing on native hardware.

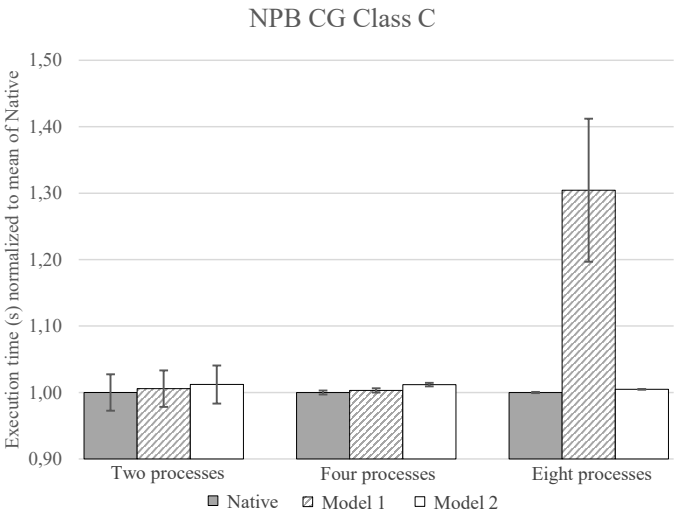

Figure 3. Native execution times from ten NPB CG runs. The Native average execution time of each CG are normalized to one and compared to container Model 1 and 2. All the two process-runs fluctuate a lot while the most stable measurements are from four processes, which have a low standard deviation. On the other hand, eight processes perform significantly worse with Model 1 than with Model 2 and has a high standard deviation, illustrated with the error bars.

Two processes native mean execution time was 168,22 seconds as shown in Table II, this value fluctuated between 164 and 176 seconds. Comparing this with the two orchestration models, a mean slowdown of one second (0,6%) and two

seconds (1,2%) was found. However, the difference, when comparing the models to each other were minimal. One second or (0,6%), and execution times fluctuate between 163 and 175 seconds for Model 1 and between 165 and 181 seconds for Model 2. Model 1 running four processes had a slower execution time of (0,3%) compared to Native while Model 2 execution time was (1,2%) slower than Native. The percentage difference between the two orchestration models was (0,9%).

TABLE II
RESULTS COMPARING DOCKER CONTAINERS TO NATIVE. THE TIME IS PRESENTED AS MEAN OF TEN INDEPENDENT ITERATIONS. THE PERCENTAGE REVEALS THE DECREASE OF TIME BETWEEN THE TWO MODELS COMPARED TO NATIVE.

	NPB	Native		Docker Model 1		Docker Model 2	
		Time (s)	St.dev.	Time (s)	St.dev.	Time (s)	St.dev.
1.	CG.C.2	168,22	4,609	169,19 (0,6%)	4,614	170,23 (1,2%)	4,818
2.	CG.C.4	100,10	0,298	100,41 (0,3%)	0,320	101,28 (1,2%)	0,264
3.	CG.C.8	71,87	0,061	93,76 (30,4%)	7,737	72,22 (0,5%)	0,038

The most distinct result, and contradictory to J. Higgins et.al. [13], was running eight processes spread over two machines. Model 2 was only 0,5% slower than native and outperformed Model 1 that was 30% slower. Model 2 had the lowest recorded standard deviation of all tests (0,038), which tells us that these are reliable numbers.

The big spike with eight processes and Model 1 could be the cause of the CG benchmark testing irregular communication and the fact that the process-launcher was scheduled within the first container of two. The first container needed to keep track of four processes and establishing TCP-connections to the other container (running four processes inside) was most likely the cause. In comparison to the spike, Model 2 running eight processes and eight containers. Then the process-launcher only had to deal with one parallel process inside the first container and multiple TCP-connections to the other seven processes, which could go via separate containers for each process. Scheduling tasks in this way thus required less communication-overhead.

V. PAUSE AND UNPAUSE

To pause running containers with parallel processes executing inside Docker engine calls the Linux cgroup freezer subsystem. Processes inside containers are automatically placed in cgroups, which the freezer system use to allocate processes to be freezed and thawed. Containers that are being paused by Docker engine places their processes in frozen states. This allows for CPU cycles to be released for other tasks, effectively freeing up some critical resources in a cluster like expensive software licenses linked to physical hardware. This has several benefits but are not without downsides. The paused container still resides in volatile RAM and will not survive a crash or reboot, if one container or part of a job is lost, then the whole job has to be restarted all over again. When unpausing containers, processes are converted to thawed states. Unfortunately no built-in function exists, at the time of this writing, to move containers to other nodes in frozen states, using the function `docker pause`.

To evaluate and find out if MPI communication and parallel processes can be frozen with `docker pause`, CG was scheduled to execute inside the pre-deployed containers according to the two orchestration models and declarations under Section IV-A. Illustrated in Figure 4, `docker pause` is called by a Python script listed in Appendix B. Containers run CG for 60 seconds, then they are put in a paused state for another 60 seconds and finally resumed, allowing executions to finish.

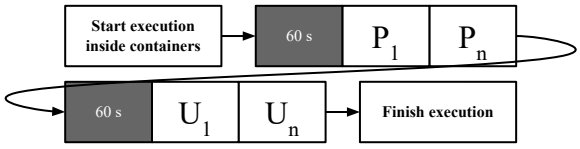

Figure 4. Flow chart for scripts executed in the Pause (P) and Unpause (U) tests. Applications started executing inside containers, and were allowed to run for 60 seconds. Containers were then paused for a duration of 60 seconds and unpaused. Finally, containers were allowed to finish execution. The numbers and n represent containers.

In Table III results showed us that running either two, four or eight processes with both container orchestration models all finished successfully. Our tests demonstrate that parallel MPI communication can be paused and unpaused, running inside Docker containers. The command `docker unpause` never returned any errors, regardless of which container orchestration model that were being tested. Extended Pause tests were then done with Model 2 and eight processes because that test was distributed over the two testbeds and Model 1 had spikes in previous performance comparisons to native hardware. All containers were paused for nine hours, emulating a general off-hour timespan, and were then unpaused. Table III also demonstrate that these tests were successfully working and indicate that it is very well likely to have simulations or virtual engineering applications, that communicate with MPI, in Docker containers paused for longer periods of time with the ability to unpause them later when resources are free.

Shuffling containers and then pausing them were next up to control if the sequence and time of Pausing (P) and Unpausing (U) mattered. Eight processes started to execute inside containers for 60 seconds, illustrated in Figure 5, before the sequences of pause were shuffled and the first container was paused. A sleep timer of 1-300 seconds in between pausing the rest of the containers were invoked. Another sleep timer of 1-120 seconds was executed before re-shuffling the sequence of unpausing containers. One container was unpaused and sleep timers of 1-300 seconds were put in between unpausing the rest. Finally, the benchmark was given time to finish executing the run. The full script that did all this is listed in Appendix C.

Found in Table III, results from shuffling the sequence and time in which containers were paused and unpaused demonstrate that this had no effect on the ability to finish an

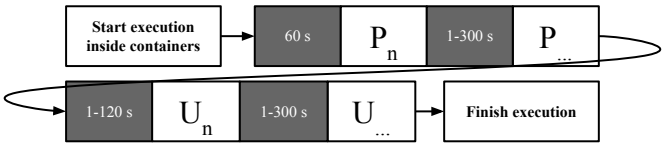

Figure 5. One iteration of controlling if the sequence and time of Pausing (P) and Unpausing (U) mattered. The n represent any shuffled container.

eight process job with Model 2. By just adding this function to a HPC cluster, jobs can persist in RAM throughout office-hours and could be resumed off-hours, with the benefit of not needing to be restarted from the beginning. However, assume that a 24 core job is spread to six compute-nodes and then paused. To unpause these jobs there has to be four cores available on each node. It might take a long time for this to happen, depending on how utilized the HPC-cluster as a whole is. Right now, in the current state, Pause is not a flexible solution to the problem of priorities, shared resources and expensive licensing costs. Jobs are locked to each node in addition to the overhead of paused jobs kept in volatile RAM, taking up hardware resources that could otherwise be used for other jobs during production hours. If one node in the cluster needs maintenance and a reboot is required, then the whole distributed simulation has to be restarted all over again. These pros and cons have to be carefully considered when just relying on the pause feature of Docker.

TABLE III
SHOWS THAT ALL DOCKER PAUSE TESTS (TWO, FOUR AND EIGHT), REPEATED TEN TIMES WORKED. EXTENDED PAUSE WERE COMPLETED WITH EIGHT PROCESSES (CG.C.8) AND WITH ORCHESTRATION MODEL 2. SHUFFLING THE ORDER OF PAUSE AND CONTROLLING TIMERS 25 TIMES ALL FINISHED SUCCESSFULLY. THE SHUFFLE SEQUENCE IS ILLUSTRATED IN FIGURE 5.

	NPB	Docker Model 1	Docker Model 2	Ext. Pause (9 h)	Shuffle Pause
1.	CG.C.2	10 of 10	10 of 10	-	-
2.	CG.C.4	10 of 10	10 of 10	-	-
3.	CG.C.8	10 of 10	10 of 10	10 of 10	25 of 25

VI. CHECKPOINT AND RESTORE (C/R)

CRIU does not support parallel/distributed computational libraries according to the official homepage [5] and M. Rodríguez-Pascual et.al. [25] points out in their personal experience that CRIU lacks the possibility to checkpoint distributed MPI-applications. CRIU is also missing the support of checkpointing Docker containers with TTY (-t) flag enabled, which was removed from all containers in the following tests.

Figure 6 presents all combinations that were based on the fact that the process-launcher was contained in the first container. Four main combinations were then derived from this fact and are as follows. Checkpoint (C) the MPI process-launcher either First and Restore (R) it First (CF/RF), or checkpoint the launcher in the opposite sequence (i.e., Checkpoint the launcher Last and Restore it Last (CL/RL). The final two main combinations are Checkpoint the launcher First and Restore it Last (CF/RL) or Checkpoint the launcher Last and Restore it First (CL/RF). Later, sub-tests of these four main combinations were also done to find out if the container-sequences before or after the launcher had an effect on the benchmark up to eight processes in eight containers. Systematically testing these main combinations always started from the least complex CG.C.2 on one testbed and ended with the most complex CG.C.8 distributed over both testbeds.

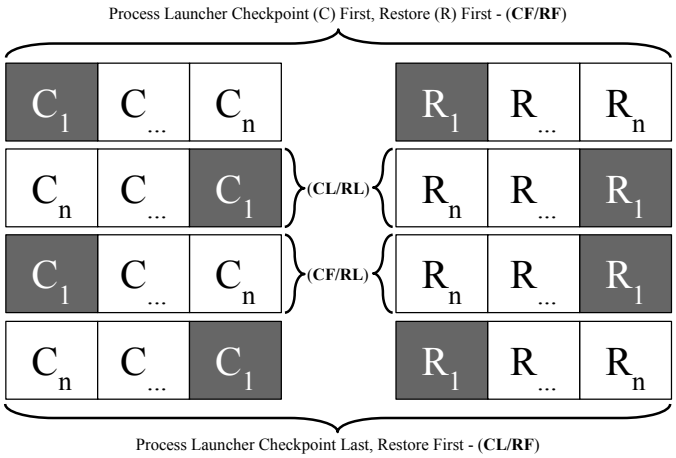

Figure 6. The MPI process launcher in all tests originated from the first container and were restored in these four main combinations. The number one illustrate the first Docker container.

All tests from here on out were started according to the declaration in Section IV-A. However, it was quickly found that Model 1 did not work with Docker C/R when several parallel processes were executing and communicating with each other inside one container. The process-launcher of MPI then returned errors while dumping processes to image files and quit executing. Therefore all consecutive tests were only using Model 2. Nothing was written by the containers to the shared Docker volume /hpc, mounted earlier, since this also returned errors and crashed CRIU while containers were mounting the volume again during restore.

A. CG.C.2 on Alpha

To begin, since it was either stated unsupported or unknown, wherever we looked, if Docker could checkpoint parallel processes communicating with MPI inside containers. A series of experiments, designed to find out the effect of timers (sleep timers) and what different combinations had on the C/R success rate. The first sets of experiments were started with least complex NPB CG.C.2 benchmark. Results from all tested combinations are found in Table IV.

B. Results and Analysis

Timers and the sequence order of C/R had an affect on successfully completing a C/R. Looking at the first set of tests in row 1-4, the third row had the best outcome without any sleep timers at all. And the bad results of the other combinations were due to containers not having enough time

TABLE IV
PRESENTS ALL CHECKPOINT/RESTORE (C/R) EXPERIMENTS THAT WERE EXECUTED ON TESTBED ALPHA. EACH COMBINATION WAS REPEATED WITH DIFFERENT SLEEP TIMERS AND WITH DIFFERENT SEQUENCES TO FIND COMBINATIONS THAT WORKED BETTER OR WORSE.

	C Sleep (s)	Checkpoint (C)	C-R Sleep (s)	Restore (R)	R Sleep (s)	Success		C. S. (s)	C.	C-R S. (s)	R.	R. S. (s)	Success		C. S. (s)	C.	C-R S. (s)	R.	R. S. (s)	Success
1.	0	**C1**, C2	0	**R1**, R2	0	1 of 10	**9.**	0	**C1**, C2	30	**R1**, R2	30	10 of 10	**17.**	0	**C1**, C2	30	**R1**, R2	1	0 of 10
2.	0	**C1**, C2	0	R2, **R1**	0	2 of 10	**10.**	0	**C1**, C2	30	R2, **R1**	30	10 of 10	**18.**	0	**C1**, C2	30	R2, **R1**	1	9 of 10
3.	0	C2, **C1**	0	R2, **R1**	0	9 of 10	**11.**	0	C2, **C1**	30	R2, **R1**	30	10 of 10	**19.**	0	C2, **C1**	30	R2, **R1**	1	9 of 10
4.	0	C2, **C1**	0	**R1**, R2	0	0 of 10	**12.**	0	C2, **C1**	30	**R1**, R2	30	10 of 10	**20.**	0	C2, **C1**	30	**R1**, R2	1	0 of 10
5.	30	**C1**, C2	30	**R1**, R2	0	0 of 10	**13.**	30	**C1**, C2	30	**R1**, R2	30	10 of 10	**21.**	0	**C1**, C2	0	**R1**, R2	30	10 of 10
6.	30	**C1**, C2	30	R2, **R1**	0	9 of 10	**14.**	30	**C1**, C2	30	R2, **R1**	30	10 of 10	**22.**	0	**C1**, C2	0	R2, **R1**	30	2 of 10
7.	30	C2, **C1**	30	R2, **R1**	0	10 of 10	**15.**	30	C2, **C1**	30	R2, **R1**	30	10 of 10	**23.**	0	C2, **C1**	0	R2, **R1**	30	10 of 10
8.	30	C2, **C1**	30	**R1**, R2	0	0 of 10	**16.**	30	C2, **C1**	30	**R1**, R2	30	10 of 10	**24.**	0	C2, **C1**	0	**R1**, R2	30	4 of 10

to even be restored at all. Only one container of the two started, the one being restored closest to its own checkpoint failed to start. The first row in the first set should get a higher success rate when comparing it against row three, if only time affected the outcome because there was a small window between restores in these two tests. However, this is not the only factor to account for, the sequence order of C/R must also be considered.

Since some containers failed to completely restore in the first set, a timer was needed in-between the last checkpoint and the first restore (C-R Sleep). This effectively eliminated the problem that containers did not start at all. CRIU and Docker were also allowed a window of 30 seconds in-between executing the command to checkpoint containers. This successfully improved both the combination CF/RL and CL/CL. The third set (row 9-12) removed the Checkpoint timer (C-Sleep) and added a Restore timer (R-Sleep). All containers and their NPB CG.C.2 finished successfully. The sequence order of C/R seemed less prominent while having the additional time between restores. The fourth set (row 13-16) further confirmed that successful C/R was possible with the main combinations while C-Sleep and R-Sleep was set to 30 seconds.

In the fifth set, which was an extension of the third set, the timer in-between restores were controlled. It did not only reveal that the two checkpoint sequence orders of CF/RL and CL/RL emerged more successful again. But it also told us that the timer in-between restores did play a part in the success of C/R and had to be somewhere in-between one second and 30 seconds. The one thing that was missing to test at this point was to remove the C-R Sleep while keeping the R-Sleep timer, which was done in the final set. When we did this, it was revealed that the two combinations that worked best was CF/RF and CL/RL that once again was proven to be the two combinations that successfully finished the benchmark the most times.

C. CG.C.4 on Alpha

By taking the combinations and timers that worked from CG.C.2 for further investigation, to the more complex CG.C.4 maxing out the four physical cores on testbed Alpha. The sequence order of checkpointing the MPI process-launcher was first set to be Checkpointed either First and Restored First (CF/RF) or CL/RL. Remaining containers shifted place to check if that had any effect on the outcome. Last but not least, the final set in Table V on row 7-8 controlled the two remaining main combinations with regards to the launcher, Checkpoint First, Restore Last (CF/RL) and CL/RF.

D. Results and Analysis

Listed in Table V, results from these experiments were all successful. While changing the order of restoring the remaining containers except the process-launcher (C1/R1) in the two first sets (row 1-3 and 4-6), showed us that this did not affect the outcome of C/R. Further, as a control, the last two main combinations were normally tested while the remaining sequence orders of containers were locked in place. This, as expected, was also completely successful. However, all containers were still located on the same testbed, which might explain why communication worked so well. MPI communication does not have to go through TCP/IP over to containers on the other testbed. At this point, given the high success rate, we moved forward to the more complex communication of NPB CG.C.8.

TABLE V
DEMONSTRATES THAT ALL COMBINATIONS TESTED WAS FOUND TO BE WORKING WHILE CHECKPOINTING AND RESTORING (C/R) THE NPB CG.C.4 BENCHMARK.

	C Sleep (S)	Checkpoint (C)	C-R Sleep (s)	Restore (R)	R Sleep (s)	Success
1.	0	**C1**, C2, C3, C4	30	**R1**, R2, R3, R4	30	10 of 10
2.	0	**C1**, C2, C3, C4	30	**R1**, R3, R2, R4	30	10 of 10
3.	0	**C1**, C2, C3, C4	30	**R1**, R4, R3, R2	30	10 of 10
4.	0	C4, C3, C2, **C1**	30	R4, R3, R2, **R1**	30	10 of 10
5.	0	C4, C3, C2, **C1**	30	R3, R2, R4, **R1**	30	10 of 10
6.	0	C4, C3, C2, **C1**	30	R2, R3, R4, **R1**	30	10 of 10
7.	0	**C1**, C2, C3, C4	30	R4, R3, R2, **R1**	30	10 of 10
8.	0	C4, C3, C2, **C1**	30	**R1**, R2, R3, R4	30	10 of 10

E. CG.C.8 on Alpha and Beta

Pre-deploying containers and distributed them over both testbed Alpha and Beta was first done. Five sets of experiments then followed which demonstrated the possibility to C/R the NPB CG.C.8, while both testbeds were under full workload. We stuck to the four main combinations in Figure 6 and extended tests that returned 100% success beyond just repeating them ten times. Those tests were repeated further, 90 times more, to find a more accurate reading of how successful those

combinations actually were. All results from CG.C.8 are listed in Table VI.

F. Results and Analysis

Interestingly, the distribution of containers over both testbeds returned mixed results, while running a larger job, the sequence order must be considered carefully. Not only is this the largest job that we will test C/R on, now TCP connections between the two testbeds also has to be restored. Communication from processes and containers themselves have to synchronize upon restoring checkpoints. Now, it was evident and more important to find a working sequence order for this to actually work in a production cluster. The low success rate of the first CF/RF combination was because of the process-launcher not being able to synchronize with the rest of the containers and processes inside upon restore. A better six of ten, but not a great result, was returned while doing the opposite CL/RL which gave the launcher a better starting position whilst restoring it, as shown in row two of Table VI.

Just reversing containers, without regards to the launcher completely failed outlined in row three which further told us that the launcher had to be carefully accounted for.

Since CF/RL and CL/RF were now the best two combinations, that first got ten of ten successful checkpoint and restores. Further testing of those two combinations followed the same pattern out of 100 runs, with the small hiccup of the CL/RF, failing to restore one time. This high success rate called for even further evaluation to control if it was possible to shuffle other containers except the process-launcher, similar to as we did with the NPB CG.C.4 previously. Two sets of four experiments in total controlled both the shuffle and reversing the shuffled order and shuffle/shuffle were done. The latter was repeated 100 times to get a better reading. It worked out to a high degree while doing it in this way but a much better result was previously found with the two main combinations CF/RL and CL/RF. If one combination had to be picked of these two, then it seems that CF/RL was the more robust of the two.

One have to note that checkpoints are still located on the same machine where the job was started, which meant that checkpoints could only be started again at the same testbed and in the same container because the checkpoint was linked to the container itself at this stage. One minor problem still exists, the flexibility to move the checkpoint to another node and restore it there.

G. C/R in New Containers

From here on out, to be able to restore an existing checkpoint in a new container, the image files had to be separated from the container-name and the container itself, which it was not before. When removing containers the checkpoints associated with the container were also removed. Luckily, Docker has the ability to specify where checkpoints are saved with the command `docker checkpoint create --checkpoint-dir=<path> <container> <checkpoint>`. The syntax to restore checkpoints are then slightly changed to `docker start --checkpoint-dir=<path> --checkpoint=<checkpoint> <container>`.

TABLE VI
LISTS ALL CG.C.8 C/R COMBINATIONS THAT WERE TESTED. SOME TESTS WERE DONE ONE HUNDRED TIMES TO DEMONSTRATE A MORE ACCURATE READING. THE PROCESS-LAUNCHER, HIGHLIGHTED WITH BOLD TEXT, WAS ALWAYS EXECUTING INSIDE THE FIRST CONTAINER.

	Seq.	Checkpoint (C)	C-R Sleep (s)	Restore (R)	R Sleep (s)	Success
1.	CF/RF	**C1**, C2, C3, C4, C5, C6, C7, C8	30	**R1**, R2, R3, R4, R5, R6, R7, R8	30	3 of 10
2.	CL/RL	C8, C7, C6, C5, C4, C3, C2, **C1**	30	R8, R7, R6, R5, R4, R3, R2, **R1**	30	6 of 10
3.	Rev.	C7, C6, **C1**, C2, C3, C4, C5, C8	30	R8, R5, R4, R3, R2, **R1**, R6, R7	30	0 of 10
4.	CF/RL	**C1**, C2, C3, C4, C5, C6, C7, C8	30	R8, R7, R6, R5, R4, R3, R2, **R1**	30	100 of 100
5.	CL/RF	C8, C7, C6, C5, C4, C3, C2, **C1**	30	**R1**, R2, R3, R4, R5, R6, R7, R8	30	99 of 100
6.	CF/RL	**C1**, Shuffle C2-C8	30	Reverse Shuffle, **R1**	30	6 of 10
7.	CL/RF	Shuffle C2-C8, **C1**	30	**R1**, Reverse Shuffle	30	9 of 10
8.	CF/RL	**C1**, Shuffle C2-C8	30	Shuffle C2-C8, **C1**	30	81 of 100
9.	CL/RF	Shuffle C2-C8, **C1**	30	**R1**, Shuffle C2-C8	30	70 of 100

Before being able to move checkpoints to a different node in a cluster, the first step was to remove containers and restoring checkpoints in new containers. New containers also had to have the same IP address as it originally had when checkpointing them on the opposite testbed. To ensure this static assignment of addresses was done. C/R in new containers was conducted by following the five steps below.

1) Start a new CG.C.8 in eight containers distributed over Alpha and Beta,
2) After a while checkpoint the job according to the combinations CF/RL or CL/RF,
3) Completely remove containers with `docker rm -f <container>`
4) `docker run`, to create new containers on both testbeds from the earlier built image in Section IV,
5) Restore checkpoints in the new containers and add the result to Table VII,
6) Repeat steps one to five ten times for each of the two main combinations.

These test differs from previous tests in the way that neither Docker nor CRIU handle the termination of containers. We also rebooted a whole testbed in between restoring containers and it had no effect on the ability to restore containers afterwards.

H. Results and Analysis

The HPC-cluster, who always strive for the best performance and optimal use of hardware resources, might want to remove containers completely when jobs have been checkpointed. For example, recreating containers and starting checkpoints can then be done only when they are needed. Checkpoints will only take up disk space and no other hardware resources at the point when containers have been removed. We demonstrate that it is possible to create containers on-demand prior to restoring checkpoints, with the two combinations as shown in Table VII. All tests returned successful C/R in new containers, which was a great foundation to build upon for the upcoming experiments.

TABLE VII
SHOW RESULTS FROM THE TWO MAIN COMBINATIONS CF/RL AND CL/RF WHILE RESTORING CHECKPOINTS IN NEW CONTAINERS RUNNING THE NPB CG.C.8.

	Checkpoint (C)	C-R Sleep (s)	Restore (R)	R Sleep (s)	Success
1.	**C1**, C2, C3, C4, C5, C6, C7, C8	30	R8, R7, R6, R5, R4, R3, R2, **R1**	30	10 of 10
2.	C8, C7, C6, C5, C4, C3, C2, **C1**	30	**R1**, R2, R3, R4, R5, R6, R7, R8	30	10 of 10

I. Summary and Overall Analysis of C/R

Summing up every test from CG.C.2 to CG.C.8 and restoring checkpoints in new containers we clearly see that time and sequence order, with regards to the process-launcher has effects on the ability to C/R parallel processes communicating from inside Docker containers. It was also quickly found that distribution according to Model 1 does not work with C/R with the current state of the experimental feature in Docker that use CRIU to do this.

Docker failed to restore containers if the last checkpointed container was restored first, without any sleep timers in between restores. There was simply not enough time for the restore to happen. This is not really a problem that would occur in a real-world scenario other than if a cluster wants to take a snapshot of a long-running job with the checkpoint feature. Then, the Docker flag `--leave-running` could be added to the checkpoint command, to have containers continue the job while creating checkpoints. This feature and the ability to do so needs to be evaluated further which it was not in this thesis.

There is native support in CRIU to control ghost-file size, but Docker lacks the option to specify a larger size than 82 MB. Studying CRIU crash logs hinted that the culprit when trying to C/R Model 1 could be the cause of this relatively low file size. It would be interesting to see what the effect a larger ghost-file limit would have on Model 1. In our small-scale tests we demonstrate that, if you carefully consider the order in which you C/R containers and use orchestration Model 2, then it is possible to work with the small ghost-file limit enforced by the experimental Docker checkpoint feature.

VII. CHECKPOINT, MOVE AND RESTORE (C/M/R)

As illustrated in Figure 7, to move a job started on Alpha to the other testbed and start it there, the job first needs to be checkpointed and saved to a shared medium. Then, the checkpoint had to be restored in containers that use the same IP addresses as the original containers had for MPI to successfully synchronize and complete. If this test is successfully demonstrated to work, then the ability to move jobs around in a cluster could be possible. If it is possible, then the great flexibility of container-virtualization can be combined with HPC, allowing jobs to move on-demand, freeing up resources for higher-prioritized jobs almost instantaneously. However, this demonstration of C/M/R was not distributed over the two testbeds at the same time. However it was yet another stepping stone to build upon for the more advanced and distributed jobs that follow in Section VIII.

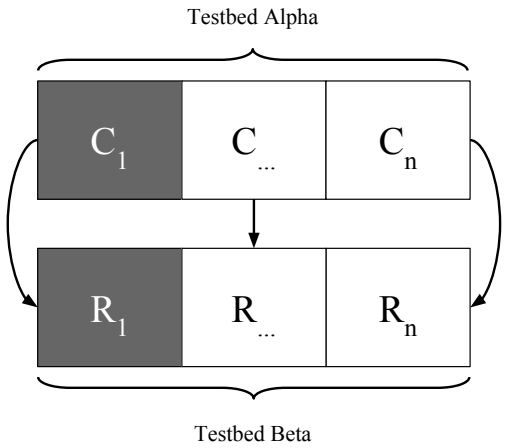

Figure 7. Shows how Checkpoints (C), were moved from one testbed to the other and then Restored (R).

A. Results and Analysis

Listed in the first two rows of Table VIII are results from CG.C.2. Since the main combination, derived from earlier tests, were both successful ten of ten times and restarting the *same* checkpoint worked. Then it was natural to move forward with the CG.C.4 that yielded much more fluctuating and unreliable results. Which in turn led us to increasing the C-R Sleep time to 120 seconds between the last checkpoint and first restore, to make sure that the shared NFS had enough time to write checkpoints to disk before they were restored on Beta. Further steps were also done to make sure that the NFS was synchronized between Alpha and Beta, where all image files of the checkpoint were stored. A Python script was leveraged to automate the C/M/R while at the same time ensured the least error prone approach to these inconsistencies. Still, there was no clear explanation why results deviated so much with the same combinations that were robust in the previous Section VI. More tests revealed that CL/RL was much more robust whilst restoring checkpoints on the opposite testbed.

The reason for CG.C.2 being so successful was probably because the two processes running on a single machine was not very complicated in comparison to CG.C.4, which is still on the same machine but more things has to synchronize upon restore. New containers were demonstrated to work on the same machine but obviously many more things have to be synchronized while restoring checkpoints on a completely new and independent machine (e.g., process IDs and its child processes, Docker network, communication between processes and so on).

When looking at the combinations that worked when restarting the checkpoint on the same machine, in earlier experiments from Table VI, and comparing them with the results found while moving the checkpoint to a different testbed in Table VIII. Results now tells us that the working combination is only CL and RL whereas before it was both CF/RL and CL/RF. At least with the CG.C.4 benchmark.

TABLE VIII
CHECKPOINTS ON TESTBED ALPHA MOVED AND RESTORED ON TESTBED BETA. TWO PROCESSES ARE CHECKPOINTED ON ROW 1-2 AND FOUR ON ROW 3-4. THE CHECKPOINT-RESTORE (C-R) SLEEP TIMER WAS INCREASED TO 120 SECONDS WHEN RUNNING FOUR PROCESSES. RETRY COLUMN REPRESENT THE PREVIOUS CHECKPOINT BEING RESTORED ONE MORE TIME.

	Seq.	C. on Alpha	C-R Sleep (s)	R. on Beta	R. Sleep (s)	Success	Retry
1.	CF/RL	**C1**, C2	30	R2, **R1**	30	10 of 10	10 of 10
2.	CL/RF	C2, **C1**	30	**R1**, R2	30	10 of 10	10 of 10
3.	CF/RL	**C1**, C2, C3, C4	120	R4, R3, R2, **R1**	30	3 of 10	3 of 10
4.	CL/RF	C4, C3, C2, **C1**	120	**R1**, R2, R3, R4	30	3 of 10	3 of 10
5.	CF/RF	**C1**, C2, C3, C4	120	**R1**, R2, R3, R4	30	0 of 10	0 of 10
6.	CL/RL	C4, C3, C2, **C1**	120	R4, R3, R2, **R1**	30	9 of 10	9 of 10

B. Further Evaluating the Inconsistent Results

To make sure that the NFS and file sync between Alpha and Beta was not the culprit of the less successful C/M/R. Completely detaching the NFS was done by copying files via Secure Copy Protocol (SCP). The bash command `sync` was also incorporated on each testbed to write any data buffered in memory to disk, eliminating possible synchronization issues between the image files copied from Alpha to Beta. The script that produced the results in Table IX was looping through the main combinations ten times and then looked for a successful restore. If no successful restore was found then the script tried to restart the *same* checkpoint ten more times. Results show that all successful restores were successfully completed on the first try, therefore it got the number one. When the benchmark failed eleven times it got an x.

TABLE IX
SECURE COPY CHECKPOINTS CREATED ON ALPHA TO BE RESTORED ON BETA. THE NUMBER ONE INDICATE A SUCCESSFUL CHECKPOINT, MOVE AND RESTORE ON THE FIRST TRY. THE X IS ELEVEN FAILED ATTEMPTS.

	Seq.	1	2	3	4	5	6	7	8	9	10
1.	CF/RL	1	1	x	x	x	x	x	x	x	x
2.	CL/RF	x	x	x	x	x	x	x	x	x	x
3.	CF/RF	1	1	1	1	1	1	1	1	1	1
4.	CL/RL	1	1	1	1	1	1	1	1	1	1

VIII. CHECKPOINT, SWITCH AND RESTORE C/S/R

In the last concluding tests, distributed parts of containers were checkpointed on opposite testbeds and moved according to the illustration in Figure 8. The process launcher had fixed hostname-to-IP mapped so static IP routes had to be changed for a successful switch to happen. Since we demonstrated that it was possible to move checkpoints and restore them on a different machine in earlier tests, the possibility for this to also work were high. Switching checkpoints would demonstrate a really flexible checkpoint solution where distributed jobs could move almost on-the-fly, like Dynamic Resource Scheduling in VMware or just change computing resources while being restored during off-hours. Jobs would no longer be locked to the same node where the job was initially started. The first two experiments were done manually with pre-configured bash lines and they were repeated ten times each. Eight distributed processes was further tested with a total of 100 iterations, all automated with a Python script executing earlier bash lines for consistency. While at the same time removing the human error part of doing these fairly complex tasks manually. Results from all runs are located in Table X.

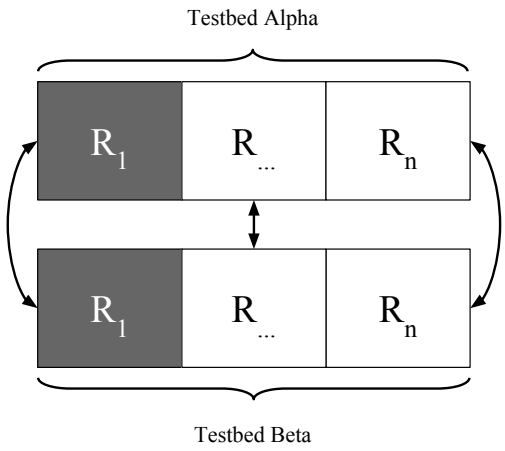

Figure 8. Shows how checkpoints, indicated by the subscripted number, were switched from one testbed to the other and then Restored (R).

A. Results and Analysis

When analyzing the last concluding experiments, two and four processes with the main combination CL/RL all returned 100% success. The most interesting CG.C.8 did not reach up to the one hundred percent success rate, which is to be expected while testing an experimental feature. However, 94% success from 100 iterations is a respectable concluding figure. Results show that some retry tests, restarting the same checkpoint again returned more finished CG.C.8 benchmarks. This could be the cause of, the now relatively low sleep timers of only 30 seconds, after the last checkpoint and before the first restore and the timer in between consecutive restores combined with the shared NFS directory in asynchronous mode. It might very well work better with faster HDDs or SSDs and we acknowledge that there still are many variables that needs to be tested further, which could affect the outcome of the results shown in Table X.

TABLE X
COMPLETELY DISTRIBUTED RUNNING CG.C.2, CG.C.4 AND CG.C.8 WERE CHECKPOINTED, SWITCHED AND RESTORED ON DIFFERENT TESTBEDS FROM WHERE THEY INITIATED. RESULTS DEMONSTRATE ONE OF THE MAIN COMBINATION CL/RL WORKING WITH HIGH SUCCESS RATES IN THESE EXPERIMENTS. THE RETRY COLUMN TESTED IF THE SAME CHECKPOINT COULD BE RESTORED AGAIN.

	C. on Alpha	C. on Beta	C-R Sleep	R. on Beta	R. on Alpha	R. Sleep (s)	Success	Retry
1.	C2	**C1**	30	R2	**R1**	30	10 of 10	10 of 10
2.	C4, C3	C2, **C1**	30	R4, R3	R2, **R1**	30	10 of 10	10 of 10
3.	C8, C7, C6, C5	C4, C3, C2, **C1**	30	R8, R7, R6, R5	R4, R3, R2, **R1**	30	94 of 100	96 of 100

IX. Discussion

With the help of Docker containers we leveraged the built-in experimental checkpoint feature to alleviate the problem with high software licensing costs tied into hardware, one of the many problems found in HPC. While performing all the experiments, several trial and error phases had to be done. We have actually never worked with, or even seen a production HPC-cluster but the good thing about Docker is the portability. Containers that worked in our test environment have a high chance of working in a production HPC-cluster as long as the cluster is running an OS that support Docker. We are both certain that in the near future, as more and more companies adopt Docker and containers, a form of live migration will emerge, containers would then directly compete with other hypervisor-based virtualization.

Developers behind Docker/CRIU have talked about a feature of first pausing containers and then checkpointing them as being under development. This combination could further increase the success rate of C/R. We tried to emulate this, by pausing containers and then checkpointing them. However, CRIU was not able to collect the processes since they were all in a frozen states. As of 2017-05-09, according to CRIUs webpage [5], Infiniband support is not available and needs to be evaluated before deciding to deploy Docker containers with the hopes of getting C/R with CRIU to work.

In our small-scale experiments we only use the NASA NPB CG benchmark, derived from computing fluid dynamics. This benchmark is not a production application but merely a mimicking simulation. There are many custom built applications running on HPC-clusters all over the world and our two, relatively old desktop-computers, hardly emulate a production HPC-cluster or the applications that actually run on them.

More experiments of everything would be a good start in validating the findings that we came up with. Testing more combinations of the distributed CG.C.8 in the concluding tests might very well reveal other working patterns and combinations that might be of interest to the industrial HPC-community. Scheduling the process-launcher within the first container was our choice. However, this can be done in many other ways and might have other outcomes on the results.

Checkpointing, moving and restoring a much larger job, on a real HPC-cluster, is far more complex for CRIU than what we can see here. Our results are however, a good stepping stone to larger implementations and applications.

X. Conclusions

Containers would alleviate problems with expensive licensing costs, that are linked to hardware-resources, while at the same time allowing for great flexibility and portability. Containers has proven to be a lightweight and cost effective alternative to common virtualization techniques, while at the same time offering modularity with process and resource isolation. In this thesis, we show that orchestration model 2 is only 0.5 percent slower than native execution time and that it is possible to Checkpoint/Restore parallel processes communicating with MPI in Docker containers. We also show that it is possible to move, checkpoints and restore them on different nodes 94 out of 100 times. Resources, time and licenses would be released if implemented. If a more reliable option is needed, that is not experimental, then we also show that there can be much to gain by using pause and resume functions in Docker compared to killing off jobs completely and restarting them from scratch, at the expense of RAM and flexibility.

Acknowledgements

We would like to thank Andreas de Blanche and anonymous readers for their thoughtful comments and feedback on this thesis.

References

[1] Docker, "Build, ship, run. an open platform for distributed applications for developers and sysadmins." Feb. 2017. [Online]. Available: https://www.docker.com/what-docker/

[2] R. Garg, K. Arya, J. Cao, G. Cooperman, J. Evans, A. Garg, N. A. Rosenberg, and K. Suresh, "Adapting the dmtcp plugin model for checkpointing of hardware emulation," *arXiv preprint arXiv:1703.00897*, 2017.

[3] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[4] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1. IOP Publishing, 2006, p. 494.

[5] "Criu." [Online]. Available: https://criu.org/Main_Page

[6] "Kvm." [Online]. Available: https://www.linux-kvm.org/page/Main_Page

[7] "Vmware virtualization for desktop and server application public and hybrid clouds." [Online]. Available: http://www.vmware.com/

[8] 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*. IEEE, 2015, pp. 171–172.

[9] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 386–393.

[10] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *Communications and Electronics (ICCE), 2016 IEEE Sixth International Conference on*. IEEE, 2016, pp. 52–57.

[11] C. L. Paul Menage, Paul Jackson, "Cgroups," Aug. 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[12] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 233–240.

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

[14] J. Meyerson, "The go programming language," *IEEE Software*, vol. 31, no. 5, pp. 104–104, 2014.

[15] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[16] "Linuxcontainers.org infrastructure for container projects." [Online]. Available: https://linuxcontainers.org/

[17] "Open mpi: Open source high performance computing." [Online]. Available: https://www.open-mpi.org/

[18] "Mpich." [Online]. Available: http://www.mpich.org/

[19] "cgroup freezer." [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt

[20] "Freezer subsystem," Jan. 2016. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[21] "Criu - checkpoint/restore in user space," Oct. 2016. [Online]. Available: https://access.redhat.com/articles/2455211
[22] Mar. 2016. [Online]. Available: https://www.nas.nasa.gov/publications/npb.html
[23] "Docker hub. centos - official repository," Mar. 2017. [Online]. Available: https://hub.docker.com/_/centos/
[24] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.
[25] M.-G. R. Rodríguez-Pascual M, Moríñigo J.A, "Checkpoint/restart in slurm: current status and new developments," September 2016. [Online]. Available: https://slurm.schedmd.com/SLUG16/ciemat-cr.pdf

Appendix A
Dockerfile

```
##############################################################################
#             Centos Docker Container for Message Passing Interface          #
#                                    -                                       #
#                      Magnus Brattlof | Gustaf Berg                         #
#        magnus.brattlof@student.hv.se | gustaf.berg@student.hv.se           #
#                                                                            #
##############################################################################

FROM centos:latest

# Download all software dependencies
RUN yum -y install openssh-server openssh-clients passwd ; yum clean all

RUN \
    # Where to locate mpi bin and lib
    echo "export PATH=/hpc/mpich/bin:$PATH" >> ~/.bashrc && \
    echo "export LD_LIBRARY_PATH=\"/hpc/mpich/lib:\$LD_LIBRARY_PATH\"" >> ~/.bashrc \
    # Configuration of SSH
    mkdir /var/run/sshd \
    sed -i 's/PermitRootLogin without-password/PermitRootLogin yes/' /etc/ssh/sshd_config \
    ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key -N '' \
    echo "pwd" | chpasswd

# Configure the host-keys
ADD ssh/config /root/.ssh/config
ADD ssh/id_rsa.mpi /root/.ssh/id_rsa
ADD ssh/id_rsa.mpi.pub /root/.ssh/id_rsa.pub
ADD ssh/id_rsa.mpi.pub /root/.ssh/authorized_keys

# Edit permissions
RUN \
    chmod -R 600 /root/.ssh/* && \
    chown -R root:root /root/.ssh/

EXPOSE 22
ENTRYPOINT ["/usr/sbin/sshd", "-D"]
ENV PATH=/usr/bin:/usr/local/bin:/bin:/app
```

Appendix B
Baseline Python Script

```
#!/usr/bin/python2.7

import subprocess as sp

experiment_list = [
 {'processes': '2', 'container': 1, 'npb': '/home/hpc/cg.C.2',
 'logfile': '/home/hpc/logs/test/test-cg2-1c.',
 'hostfile': '/home/hpc/hostfiles/2p_1c'
 },
 {'processes': '2', 'container': 2, 'npb': '/home/hpc/cg.C.2',
 'logfile': '/home/hpc/logs/test/test-cg2-2c.',
 'hostfile': '/home/hpc/hostfiles/2p_2c'
 },
 {'processes': '4', 'container': 1, 'npb': '/home/hpc/cg.C.4',
```

```
 'logfile': '/home/hpc/logs/test/test-cg4-1c.',
 'hostfile': '/home/hpc/hostfiles/4p_1c'
 },
 {'processes': '4', 'container': 4, 'npb': '/home/hpc/cg.C.4',
 'logfile': '/home/hpc/logs/test/test-cg4-4c.',
 'hostfile': '/home/hpc/hostfiles/4p_4c'
 },
 {'processes': '8', 'container': 2, 'npb': '/home/hpc/cg.C.8',
 'logfile': '/home/hpc/logs/test/test-cg8-2c.',
 'hostfile': '/home/hpc/hostfiles/8p_2c'
 },
 {'processes': '8', 'container': 8, 'npb': '/home/hpc/cg.C.8',
 'logfile': '/home/hpc/logs/test/test-cg8-8c.',
 'hostfile': '/home/hpc/hostfiles/8p_8c'
 }

]
for experiment in experiment_list:
 print "On {} processes with {} containers:".format(experiment\
 ['processes'], experiment['container'])
  for loop in range(1, 11):
   print "{} of 10 finished".format(loop)
    process = sp.Popen(['mpirun', '-outfile-pattern', experiment\
    ['logfile'] + '{0}'.format(loop),
     '-f', experiment['hostfile'], '-n', experiment['processes'],\
     experiment['npb']], shell=False)
      process.communicate()
       if loop == 10:
        print "{} processes with {} containers finished\
        successfully".format(experiment['processes'], experiment['container'])
```

APPENDIX C
EXTENDED PAUSE PYTHON SCRIPT

```
#!/usr/bin/python2.7
import docker
import time
import random
import os
import logging
import subprocess as sp
from random import shuffle


FNULL = open(os.devnull, 'w')
client = docker.APIClient(
    base_url='unix://var/run/docker.sock')
containers = ['cont01', 'cont02', 'cont03',
              'cont04', 'cont05', 'cont06', 'cont07', 'cont08']
logging.basicConfig(
    filename='/home/hpc/logs/extended_pause.log', level=logging.INFO)


def main():
    for i in range(1, 101):
        logging.info('\nOn {} iteration of 100'.format(i))
        process = sp.Popen(['ssh', 'cont01', 'mpirun', '-outfile-pattern',
```

```
                                '/hpc/logs/extended_pause/logfile.' +
                                '{}'.format(i), '-f', '/hpc/hostfiles/8p_8c', '-n',
                                '8', '/hpc/cg.C.8'], shell=False)
        time.sleep(60)
        pause()
        time.sleep(random.randrange(1, 120))
        unpause()
        process.communicate()
        time.sleep(60)


def pause():
    shuffle(containers)
    for container in containers:
        try:
            client.pause(container)
        except docker.errors.NotFound:
            sp.Popen(['ssh', 'guma02', 'docker', 'pause', container],
                     shell=False, stdout=FNULL, stderr=sp.STDOUT)
        random_sleep = random.randrange(1, 300)
        logging.info('\nPaused container: {}\nSleeping {} \
        seconds before next container'.format(
            container, random_sleep))
        time.sleep(random_sleep)


def unpause():
    shuffle(containers)
    for container in containers:
        try:
            client.unpause(container)
        except docker.errors.NotFound:
            sp.Popen(['ssh', 'guma02', 'docker', 'unpause', container],
                     shell=False, stdout=FNULL, stderr=sp.STDOUT)
        random_sleep = random.randrange(1, 300)
        logging.info('\nPaused container: {}\nSleeping {} \
        seconds before next container'.format(
            container, random_sleep))
        time.sleep(random_sleep)


if __name__ == '__main__':
    main()
```

APPENDIX D
CHECKPOINT PYTHON SCRIPT

```
#!/usr/bin/python2.7
from random import shuffle
import time
import os

def main():
    global containers, local_containers
    local_containers = ['cont01', 'cont02', 'cont03', 'cont04']
    for i in range(1,11):
        print 'On iteration', i
```

```
        containers = ['cont07', 'cont06', 'cont01',
        'cont02', 'cont03', 'cont04', 'cont05', 'cont08']

        # Remove checkpoints
        remove()

        # Restart containers
        restart()
        time.sleep(2)

        # Start MPI within container 01
        print 'Started MPI within cont01'
        os.system("ssh cont01 'mpirun -n 8 -f /hpc/hostfiles/extpause
        /hpc/cg.C.8 >> /tmp/yum.log & exit'")

        # Checkpoint containers
        time.sleep(30)
        checkpoint()
        time.sleep(30)

        # Reverse the order
        containers = reversed(containers)

        # Restore containers one by one
        restore()
        time.sleep(100)

def checkpoint():
    for container in containers:
        if container in local_containers:
            os.system('docker checkpoint create {} c1 > /dev/null'
            .format(container))
        else:
            os.system("ssh guma02 'docker checkpoint create {} c1 > /dev/null'"
            .format(container))
        print 'Checkpointed container:', container

def restore():
    for container in containers:
        if container in local_containers:
            os.system('docker start --checkpoint c1 {}'
            .format(container))
            time.sleep(30)
        else:
            os.system("ssh guma02 'docker start --checkpoint c1 {}'"
            .format(container))
            time.sleep(30)
        print 'Restored container:', container

def remove():
    print 'Removing checkpoints'
    for container in containers:
        if container in local_containers:
            os.system('docker checkpoint rm {} c1'
            .format(container))
        else:
```

```
            os.system("ssh guma02 'docker checkpoint rm {} c1'"
            .format(container))

def restart():
    print 'Restarting checkpoints'
    for container in containers:
        if container in local_containers:
            os.system('docker restart {} > /dev/null'
            .format(container))
        else:
            os.system("ssh guma02 'docker restart {} > /dev/null'".format(container))

if __name__ == '__main__':
    main()
```