D12.6 – Benchmarking results

Deliverable No.:	D12.6
Deliverable Name:	Benchmarking results
Contractual Submission Date:	31/10/2019
Actual Submission Date:	01/11/2019
Version:	V1.2

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 730897.

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**HPC-Europa3**
Project Full Name:	Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing
Deliverable No.:	D12.6
Document name:	Benchmarking results
Nature (R, P, D, O):	R
Dissemination Level (PU, PP, RE, CO):	PU
Version:	V1.2
Actual Submission Date:	01/11/2019
Author, Institution: E-Mail:	Dimitris Dellis (GRNET) ntell@grnet.gr
Other contributors	Raül Sirvent, Marta Garcia, Oleksandr Rudyy (BSC) Niall Wilson (NUIG-ICHEC) Giuseppa Muscianisi, Simone Marocchi (CINECA) Alexey Cheptsov (HLRS) Sébastien Gadrat, Guillaume Cochard (CNRS)

ABSTRACT:

The aim of this deliverable is to provide measurements of performance of selected applications that explore various subsystems and compare the performance on bare metal and containerized applications. A suite of representative applications was created. These cover various scientific fields and explore various subsystems. Benchmark results and conclusions per application are presented and discussed.

KEYWORD LIST:

High Performance Computing, HPC, Virtualization, Containers, Docker, Singularity, Shifter, MPI, GPU, SLURM, Grid Engine.

MODIFICATION CONTROL			
Version	**Date**	**Status**	**Author**
1.0	16/10/2019	Draft	D. Dellis (GRNET)
1.1	30/10/2019	Draft	D. Dellis (GRNET)
1.2	31/10/2019	Final	D. Dellis (GRNET)

The author is solely responsible for its content, it does not represent the opinion of the European Community and the Community is not responsible for any use that might be made of data appearing therein.

TABLE OF CONTENTS

1. List of Figures

2. List of Tables

3. List of Abbreviations

ALE	Arbitrary Lagrangian Eulerian
AVX2	Advanced Vector Extensions 2
AVX	Advanced Vector Extensions
BSC	Barcelona Supercomputing Center, Spain
CFD	Computational Fluid Dynamics
CINECA	CINECA Consorzio Interuniversitario, Italy
CNRS	Centre National de la Recherche Scientifique, France
CPU	Central Processing Unit
CSM	Computational Solid Mechanics
CUDA	Compute Unified Device Architecture (NVIDIA)
FMA	Fused Multiply Add
FSI	Fluid Structure Interaction
GFlops	Giga (=10_9) Floating point operations per second
GRNET	Greek Research and Technology Network S.A., Greece
HLRS	High Performance Computing Center Stuttgart, Germany
HPC	High Performance Computing
ICHEC	Irish Centre for High End Computing, Ireland
ICTP	International Centre for Theoretical Physics
LBM	Lattice Boltzmanm Method
MD	Molecular Dynamics
ML	Machine Learning
MPI	Message Passing Interface
MPMD	Multiple Program Multiple Data
OS	Operating System
PRACE	Partnership for Advanced Computing in Europe
QE	Quantum Espresso
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
SIMD	Singe Instruction Multiple Data
SSE2	Streaming SIMD Extensions 2
TCP	Transmission Control Protocol
UEABS	Unified European Applications Benchmark Suite
WLCG	Worldwide Large Hadron Collider Computing Grid
WRF	Weather Research and Forecasting

4. Executive summary

Containers are considered as an alternative for application portability among HPC centres. They offer a layer for portable execution of applications on various centres, with the question of performance penalty they may introduce. A number of representative and commonly used applications on HPC systems was selected and their behaviour with or without containers investigated. Two main ways of using containers were identified.

- A portable image that runs everywhere, at least on the same architecture (like x86 and derivatives).
- A portable development platform that is used on any machine of the same architecture in order to avoid unsatisfied dependencies on packages and versions.

From the end user point of view, the first seems to be the desirable. But, taking into account performance issues, probably the second is the best choice. On the other hand, as it will be indicated in the next sections, using containers as a development platform may involve issues that the end user does not face on bare metal. At the end, an application-based performance modelling of containerized versus bare metal is presented.

In some cases, different container frameworks were tested with the same application to investigate the possible influence of the framework on performance.

5. Introduction

In order to investigate the effect of containerization in performance and portability among different HPC systems, a number of representative applications that explore different features of HPC systems have been selected. These include CPU/GPU features, network bandwidth and latency, memory bandwidth and IO performance. At the early stage of project, a number of applications was considered and a final decision about the applications to benchmark was agreed. These applications together with the responsible site and subsystems they depend on (based on experience) are summarized in Table 3.1.

Table 5.1: Applications selected for benchmarking, corresponding site to perform benchmark and factors that affect their performance.

Site	Application	What mainly affects its performance
GRNET	GROMACS	CPU/GPU features/speed, network speed/latency
ICHEC	WRF	CPU, network, IO bandwidth and latency
BSC	ALYA	CPU, network, Memory bandwidth and latency
BSC	BT-MZ	CPU, network, Memory bandwidth and latency
BSC	HPCG	Memory
CINECA	Tensorflow (without GPU)	CPU
CINECA	Quantum Espresso	CPU, Network
HLRS	OpenFOAM	Memory, IO, CPU
HLRS	PACE3D	Memory, CPU
HLRS	Ping Pong	Network Bandwidth and Latency
HLRS	Palabos	Memory, CPU Features
HLRS	Data Analytics Workflow	IO, Memory bandwidth
CNRS	DIRAC benchmark	CPU
CNRS	RAMP Astro benchmark	GPU (CPU), Memory, I/O bandwidth

Each site carried out mainly on its own premises benchmarks and reports the obtained performance of containerized application with respect the bare metal performance. We should note that a rather large number of architectures and container frameworks were covered by these benchmarks. In addition, various frameworks for containerization were considered, based on the availability of container's frameworks at each site. Various aspects related to containers usage are covered in the following sections, namely, the effect on performance of a fully portable container image, and the issues that an end user may face among different systems with different hardware and software. Details on how the available container frameworks were used on each system are presented. Finally, after presentation and interpretation of results of each code, a general conclusion on the use of containers on an HPC environment is presented.

6. GRNET Benchmarks

GRNET selected GROMACS as application to benchmark containers versus bare metal. GROMACS is popular in studies of bio-molecular systems and materials science.

4.1. GROMACS

GROMACS [1] is a versatile package to perform molecular dynamics, i.e. simulate the Newtonian equations of motion for systems with hundreds to millions of particles. It is primarily designed for biochemical molecules like proteins, lipids and nucleic acids that have a lot of complicated bonded interactions, but since GROMACS is extremely fast at calculating the non-bonded interactions (that usually dominate simulations) many groups are also using it for research on non-biological systems, e.g. polymers. GROMACS supports all the usual algorithms you expect from a modern molecular dynamics implementation, but there are also quite a few features that make it stand out from the competition. GROMACS provides extremely high performance compared to all other programs. A lot of algorithmic optimizations have been introduced in the code; for instance, extracted the calculation of the virial from the innermost loops over pairwise interactions, and it uses its own software routines to calculate the inverse square root. In versions 4.6 and up of GROMACS, on almost all common computing platforms, the innermost loops are written in C using intrinsic functions that the compiler transforms to SIMD machine instructions, to utilize the available instruction-level parallelism. These kernels are available in either single and double precision, and in support, all the different kinds of SIMD support found in x86-family (and other) processors. It supports single node parallelization using OpenMP, multi-node using hybrid MPI - OpenMP. In addition, all these parallelization schemes have excellent CUDA-based GPU acceleration on GPUs that have NVIDIA compute capability >= 2.0.

4.2. Benchmarks Environment

GROMACS benchmarks performed on GRNET ARIS Tier-1 system. Two types of nodes were used. One part of the cluster consisting of 3 nodes of four socket E5-4650 10C v2 @ 2.40 GHz that was dedicated for HPCE3 T12.5 Benchmarks, supporting both Docker (docker-ce-18.06.0) and Singularity (v3.2.1). Since Singularity is supported cluster wide on GRNET’s ARIS System, Singularity runs were performed also on GPU nodes that are dual socket E5-2660 10C v3 with dual NVIDIA K40m GPUs.

For container images of both Docker and Singularity, a Debian 9 image was created with the necessary dependencies. For both Docker and Singularity, the host MPI and CUDA environments were mounted during compilations and runs. All runs were performed under SLURM workload manager.

The input dataset is the Case A of PRACE UEABS Benchmark Suite [2], which includes ~130,000 atoms. This size of MD system seems to be the size of the majority of GROMACS runs on an HPC Centre. Version 2018.4 of GROMACS, that was the latest stable during benchmarks, was used in all runs.

4.3. Compilation procedure

GROMACS typically selects the best SIMD instruction set to use at compile time on the compile machine. CPU features are detected automatically. It also gives the opportunity to enforce certain type of SIMD instructions for use or simply use the C/C++ kernels. This way one can emulate the compilation on another CPU type. Binary compiled at a level of SIMD instructions do not run on older CPUs that do not support it. On the other hand, binary is compatible with newer CPUs but with performance degradation. This is discussed in results section.

In order to build a fully portable image that runs on any x86_64 CPU and use it everywhere without re-compilation, the C kernels, with compiler “best effort flags” should be used. Taking into account the evolution of CPU features of x86 architecture, SSE2 is available to all x86 CPUs (both Intel and AMD) during at least last decade. So instead of plain C/C++ kernels, the SSE2 SIMD set could be used as reference.

For the compilation, the image gnu compilers were used. For Debian 9 image, they are gnu-6.3.0. The host compiler for bare metal compilation was gnu-6.4.0. For MPI and CUDA, host openmpi/2.1.6 and cuda/9.2 were mounted in image, and path/library path were adjusted accordingly. For each type of SIMD instructions set and parallelization model (OpenMP, Hybrid MPI/OpenMP/CUDA) a different executable was created. These executables were used during containerized runs.

We should note that an executable created for a certain SIMD set runs on newer CPUS that usually support older SIMD sets. On the other hand, an executable created on a recent CPU with auto CPU detection, cannot run on older CPUs. The performance is measured in [ns/day] units, that is an expression of the rate of execution, which is very similar to GFLOPS reported by Linpack [3].

All runs were performed under SLURM workload manager. The commands in SLURM script are:

Single Node runs

```
singularity exec debian9.simg Runs/runsmpdebian.sh

runsmpdebian.sh
#!/bin/sh
for i in 40 20 10 8 4 2 1; do \
export OMP_NUM_THREADS=$i
../../Packages/Singularity/debian9-gcc630/C/bin/gmx_AVX2  mdrun \
      -s topol.tpr –deffnm\
      haswell.debian9.AVX2.OMP.$OMP_NUM_THREADS\
      -cpt 10000 -ntomp $OMP_NUM_THREADS -nice 0 -nsteps 10000\
      -noconfout
done;

MPI + CUDA runs

mpirun singularity exec -B /apps/parallel/openmpi/2.1.6/gnu\
-B /apps/compilers/cuda/10.1.168 \
../../debian9.simg  \
../../Packages/Singularity/debian9-gcc630/bin/gmx_AVX2_mpi_cuda \
mdrun -s topol.tpr -deffnm \
AVX2.MPI.cuda.gpus.2.Nodes.$SLURM_NNODES.Tasks.$SLURM_NTASKS \
-cpt  10000  -nice  0  -nsteps  10000  -noconfout  -gpu_id  01  -ntomp
$OMP_NUM_THREADS
```

We should note that in the above commands there is no –np or –hostfile specification for mpirun due to the fact that these variables are handled internally by SLURM.

4.4. Performance Results

4.4.1. Single Core Performance

Runs using Bare Metal, Docker and Singularity for all types of SIMD sets that are supported by the corresponding nodes were performed in serial. In fact, the OpenMP executables were used specifying OMP_NUM_THREADS=1 and the corresponding GROMACS command line argument for OpenMP threads. Single core performance is indicative of the relative performance and probable overhead resulting from the container framework, without any implication of the container on OpenMP. The single core results are presented in Figure 4.1, for Plain C, SSE2 and AVX_256 kernels. Two main conclusions arise from Figure 4.1.

- If the same SIMD set is used, there is no essential difference in performance between Bare Metal and containerized version of GROMACS. Test machines where both Docker and Singularity are available, support AVX_256 SIMD set - this is why results are presented up to this SIMD instructions set.
- There is significant performance loss when using older SIMD (or plain C) kernels. In these benchmarks, the relative performance of GROMACS using C, SSE2 and AVX_256 is 1, 4.9, and 6.1 respectively. This means that the SSE2 is 4.9 times faster than C kernels, and

AVX_256 is 1.25 times faster than SSE2 or 6.1 times faster than plain C kernels. Since the performance of serial runs is not affected by container, the relative execution speed is this of Figure 4.1.

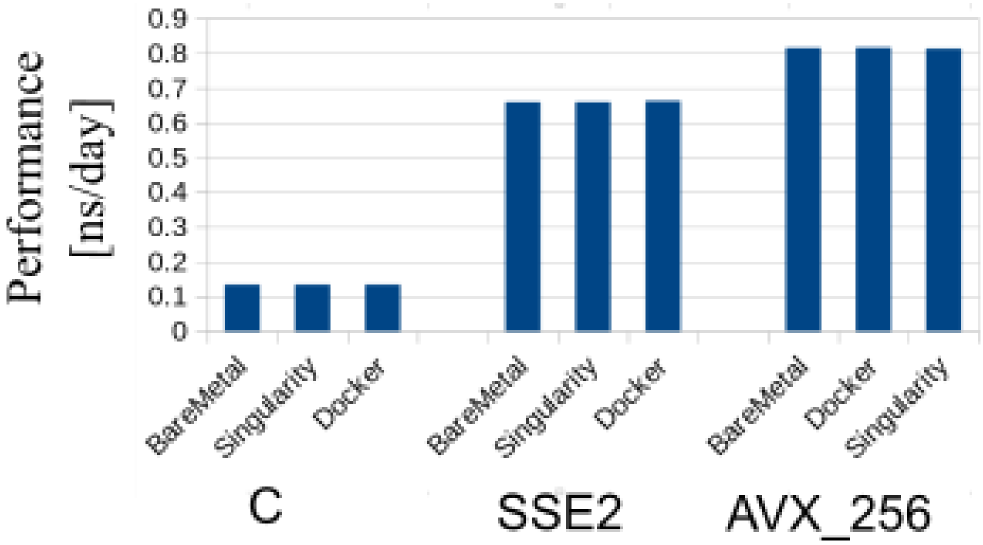


Figure 6.1: Performance of GROMACS for test case A of PRACE UEABS [2], using Bare Metal, containerized with Docker and Singularity for three different SIMD versions. Runs on four sockets node (Ivybridge E5-4650 v2 @ 2.40GHz).

Going further, the same runs were performed on a more recent CPU supporting AVX2/FMA, but Docker was not available there – only Singularity. These results are presented in Figure 4.2. From this figure it seems that the relative performance is 1 : 4.8 : 6.1 : 7.8 in the corresponding columns, that is AVX2 kernels are 1.6 times faster than SSE2. Again, if AVX2 kernels are used within container there is no essential difference in performance when compared to bare metal.

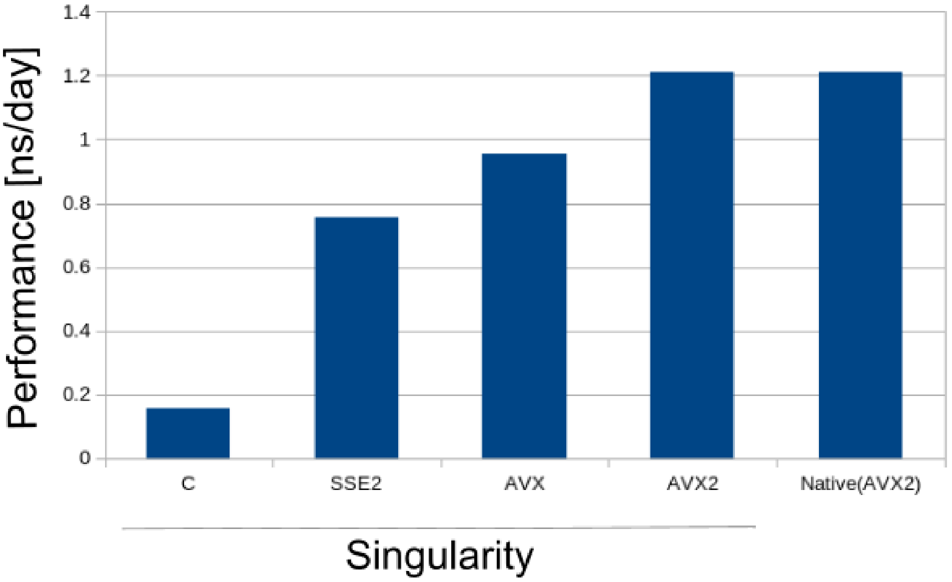


Figure 6.2: Performance of GROMACS for test case A of PRACE UEABS [2], using Bare Metal and containerized with Singularity, for four different SIMD versions. Runs on Haswell dual socket E5-2660v3 @ 2.60GHz.

4.4.2. Single Node / SMP Performance

The next level of parallelization of GROMACS is the single node loop parallelization using OpenMP. Bare metal, containerized with Docker and Singularity runs were performed on a single four socket E5-4650 v2 @ 2.40GHz node using various number of OpenMP threads and the higher available SIMD set of CPUs (AVX_256). The performance results are presented in Figure 4.3. It seems that the performance is almost identical in all cases, although very small and not systematic deviations are present.

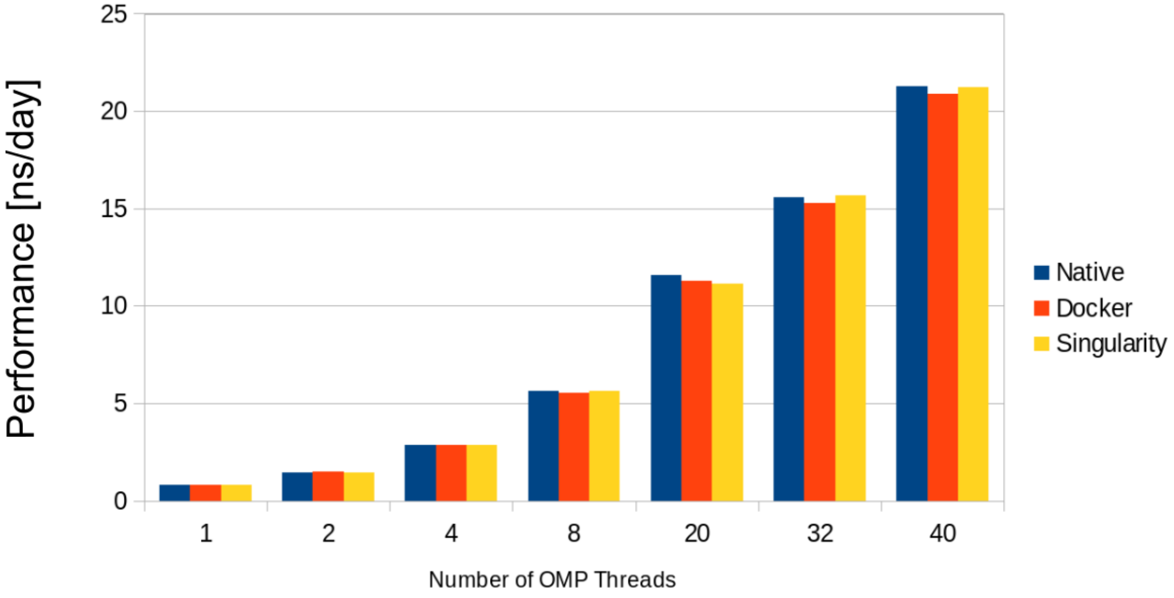


Figure 6.3: Performance of GROMACS for test case A of PRACE UEABS, using Bare Metal, containerized with Docker and Singularity for the best possible SIMD instructions as function of OpenMP threads. Runs on four sockets node (Ivybridge E5-4650 v2 @ 2.40GHz).

4.4.3. Multi node Performance

GROMACS uses MPI for the next level of parallelization. Of course, pure MPI can be used also on a single node. Communication is affected mainly by latency, while bandwidth has lower effect in scaling. For bare metal runs, the host MPI was used that leverages infiniband with RDMA for internode communication and shared memory for intranode communication. For Singularity runs, the host MPI was mounted and used in container. Although it is the same installation of MPI there are some differences. The host MPI has all driver libraries for infiniband at various places, while in container these drivers are not present. Although one can install these drivers too, it is not common for the end user to obtain, install, and probably configure these drivers. In addition, on another system other fabric drivers or versions may be used that makes this procedure difficult if not impossible to end users. In this case, the TCP transport through infiniband interface was automatically selected by openmpi for internode communication while shared memory was selected for intranode communication.

Due to the small number of available nodes to run Docker, only bare metal and singularity runs were performed with MPI. For these runs the AVX2/FMA SIMD set was used. The performance results are presented in Figure 4.4. Note that on each node 20 MPI tasks were used. From Figure 4.4, it seems that the performance with one node is essentially the same between bare metal and singularity with singularity having marginally higher performance. When more than one nodes are used, the containerized runs have a speed up to eight nodes, but their performance is significantly lower than bare metal runs. With 16 nodes, bare metal runs are 4 times faster. This is attributed to the use of TCP transport for internode communication instead of the RDMA used in bare metal.

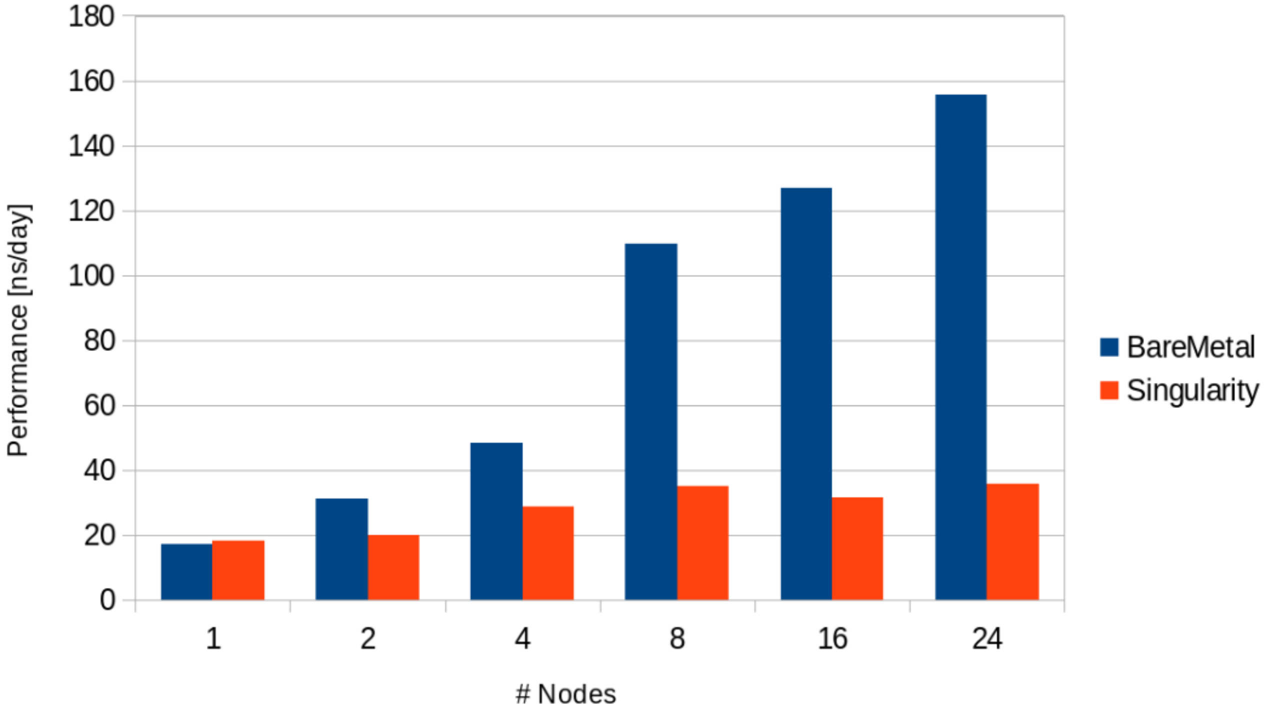


Figure 6.4: Performance of GROMACS for test case A of PRACE UEABS, using Bare Metal, containerized with Singularity for the best possible SIMD instructions (AVX2) as function of Number of Nodes. Runs on Haswell dual socket E5-2660v3 @ 2.60GHz.

4.4.4. Multi node - Multi GPU Performance

GROMACS supports also the use of GPUs with CUDA. It supports multiple GPUs per node. Since the GPU nodes have two NVIDIA K40m per node, the multi-level parallelization scheme was used: two MPI tasks per node, each task using ten OpenMP threads and one GPU. It is expected that the efficient scaling of GROMACS when using GPUs will be limited to lower number of nodes due to the GPU speed up of each MPI task. The results of these runs are presented in Figure 4.5. With number of nodes up to eight, the bare metal runs are 1.5 - 1.8 times faster than the Singularity runs. This is attributed to the effect of MPI using the TCP transport for internode communication.

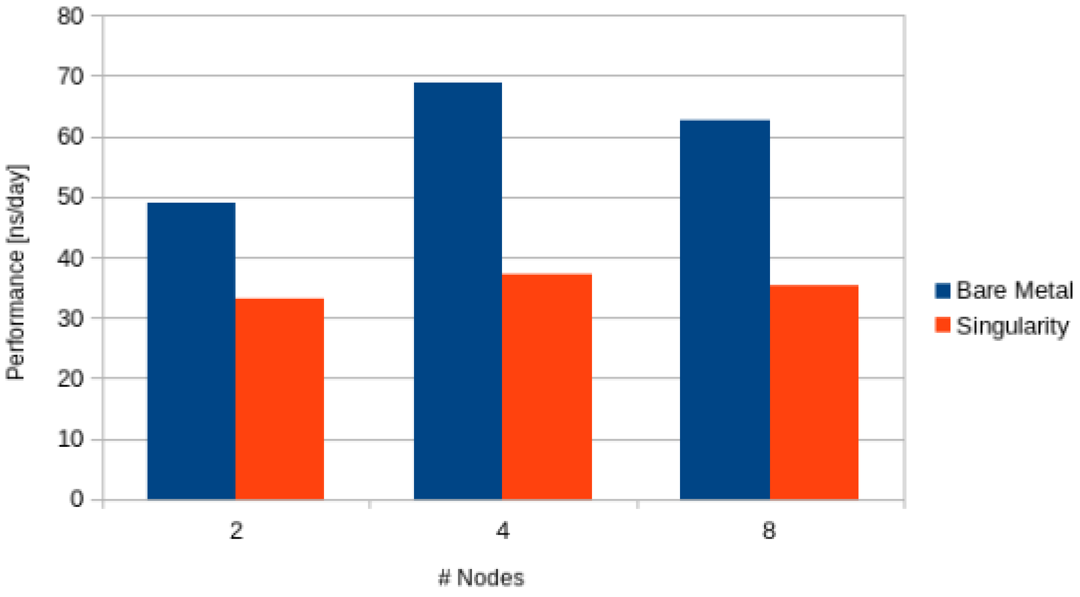


Figure 6.5: Performance of Hybrid MPI/OpenMP/CUDA GROMACS for test case A of PRACE UEABS, using Bare Metal, containerized with Singularity for the best possible SIMD instructions (AVX2) and GPUs as function of Number of Nodes. Runs on Haswell dual socket E5-2660v3.

4.5. Conclusions

Two main ways of using the containers are identified. The first, that most users have in mind, is to have a single image running as is everywhere. The second is to use the image as a deployment platform that contains all the dependencies in the required version and compile the source for the target machine with the necessary compiler optimizations/SIMD kernels to reach the maximum performance. The single image running everywhere approach implies that one should use SIMD set (in kernels and compiler flags) that is portable across available HPC machines. At present, this is the SSE2 set. This lower optimization/SIMD set has a performance penalty that might be 62% of the bare metal performance in the case of running SSE2 executable on an AVX2/FMA machine. The performance penalty is expected to be higher when running on recent skylake CPUs. In addition, if the image is built on high end CPUs, it simply doesn’t run on older CPUs.

When going to multi-node MPI runs, the performance tuning is more complicated. The single image running everywhere approach cannot be used unless one has the same versions of MPI libraries and related fabric drivers. Typically, in this case one needs to recompile the code with the available MPI on target machine and probably install various drivers in container image, still facing performance degradation and limited scalability.

Concluding, containers could be used efficiently mainly for single node runs. When containers are used as a development platform and the time-consuming code is compiled for target CPU the obtained performance is identical (or in some cases even higher) to this of bare metal case. In the case of multi-node execution, it seems that a fully portable image without effort at low level (drivers, settings etc.) introduces performance penalty that can’t commonly be handled by the end user. By the end user perspective, it seems that single node runs are fine with containers. When moving to multi-node runs, it seems for the end user, that deeper knowledge and expertise is required to achieve bare metal performance with containers.

5. ICHEC Benchmarks

5.1. WRF

The Weather Research and Forecasting (WRF) [4] Model is a mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting applications and is maintained by the National Centre for Atmospheric Research in the U.S.

Its build system is a set of custom scripts which combined with a lot of dependencies on various libraries means that it can be difficult to compile and run successfully. This build difficulty is one reason why creating a container image of a particular version may be useful. In order to investigate performance differences between native builds and containerised builds, we will build the same version of WRF using identical compilers. MPI implementations and supporting libraries to be run on both bare metal nodes and via Singularity.

5.2. Benchmarking System

All benchmark runs were performed on the ICHEC HPC system “Kay”. This system has various partitions but for WRF we only used the main distributed memory cluster partition. The relevant specifications of this cluster are:

- Each node contains 2 x 20-core 2.4 GHz Intel Xeon Gold 6148 (Skylake) processors, 192 GB of RAM
- Interconnect is Intel Omnipath 100Gbit/s RDMA network
- Lustre distributed filesystem on DDN SFA 14k hardware
- Slurm Resource Manager
- Centos 7
- Singularity 2.6

5.3. Benchmark Test Case

The purpose of this benchmark is not to examine the performance of the application or hardware itself but rather to examine the relative performance of running it under the two different scenarios (i.e. with and without containers) on the same hardware. Hence, the test case used was one of the standard tests which comes with the source distribution of WRF, namely the Tropical Cyclone. The input files are contained in the directory WRF/test/em_tropical_cyclone and to run the benchmark we just need to initialise this test case:

```
$ ../../run/ideal.exe
```

The main WRF run can then be executed as:

```
$ ln -s ../../run/wrf.exe wrf.exe
$ mpirun ./wrf.exe
```

5.4. Build and Execution of Native, “Bare Metal” version

In order to replicate as much as possible the full software stack used both natively and in the Singularity image, we used EasyBuild [5] for both to provide all the dependencies required to build WRF. In particular, the same versions of gcc, netCDF, OpenMPI, etc could be replicated this way. The following commands were used to install EasyBuild using its own bootstrapping script and then use EasyBuild to install netCDF along with all of its dependencies which contain the full environment required to build WRF.

```
$ python bootstrap_eb.py /ichec/home/staff/nwilson/work/easybuild
$ export MODULEPATH=/ichec/home/staff/nwilson/work/easybuild/modules/all
$ export EASYBUILD_INSTALLPATH=/ichec/home/staff/nwilson/work/easybuild
$ export EASYBUILD_MODULES_TOOL=EnvironmentModulesC
$ eb netCDF-Fortran-4.4.4-foss-2018b.eb --robot --module-syntax=Tcl
```

Once EasyBuild has installed all of the necessary dependencies we can then load the corresponding environment modules and compile WRF using these packages:

```
$ export MODULEPATH=/ichec/home/staff/nwilson/work/easybuild/modules/all
$ module load netCDF-Fortran/4.4.4-foss-2018b
$ export NETCDF_classic=1
$ export NETCDF=/ichec/home/staff/nwilson/work/easybuild/software/netCDF-
Fortran/4.4.4-foss-2018b
$ wget http://www2.mmm.ucar.edu/wrf/src/WRFV4.0.TAR.gz
$ tar xzf WRFV4.0.TAR.gz
$ cd WRF && ./configure <<< $'34\r1\r'
$ /bin/csh ./compile em_fire
```

The em_tropical_cyclone test case can then be run as outlined in the section above.

5.5. Build and Execution of Singularity version

For details on how the Singularity image was created using EasyBuild, please refer to Chapter 4 - Image Workflow and Usage - of Deliverable 12.2 “Container-as-a-service Technical Documentation”. This image can be used easily to run on a single node but for MPI runs on multiple nodes, we need to have the container access external MPI libraries so it can use the OmniPath libraries and Slurm task launching. In this case we use the same EasyBuild version of OpenMPI used for the bare metal runs via bind mounting those directories into the container as follows:

```
mpirun singularity exec -B
/lib64,/ichec/home/staff/nwilson/work/easybuild/software:/opt/apps/easybu
ild/software /ichec/work/staff/nwilson/singularity_images/eb-wrf.img
/opt/apps/WRF/main/wrf.exe
```

5.6. Results of Benchmark Runs

The figure below shows the execution time in seconds for both Singularity and bare metal runs scaling from 1 to 8 nodes. Docker was not tested because it was not available on the cluster. Again, the point of interest here is not the absolute performance but rather the relative performance between both configurations. As can be clearly seen, for this particular test case there is a negligible performance impact for running WRF via a Singularity container compared with native, bare metal performance.

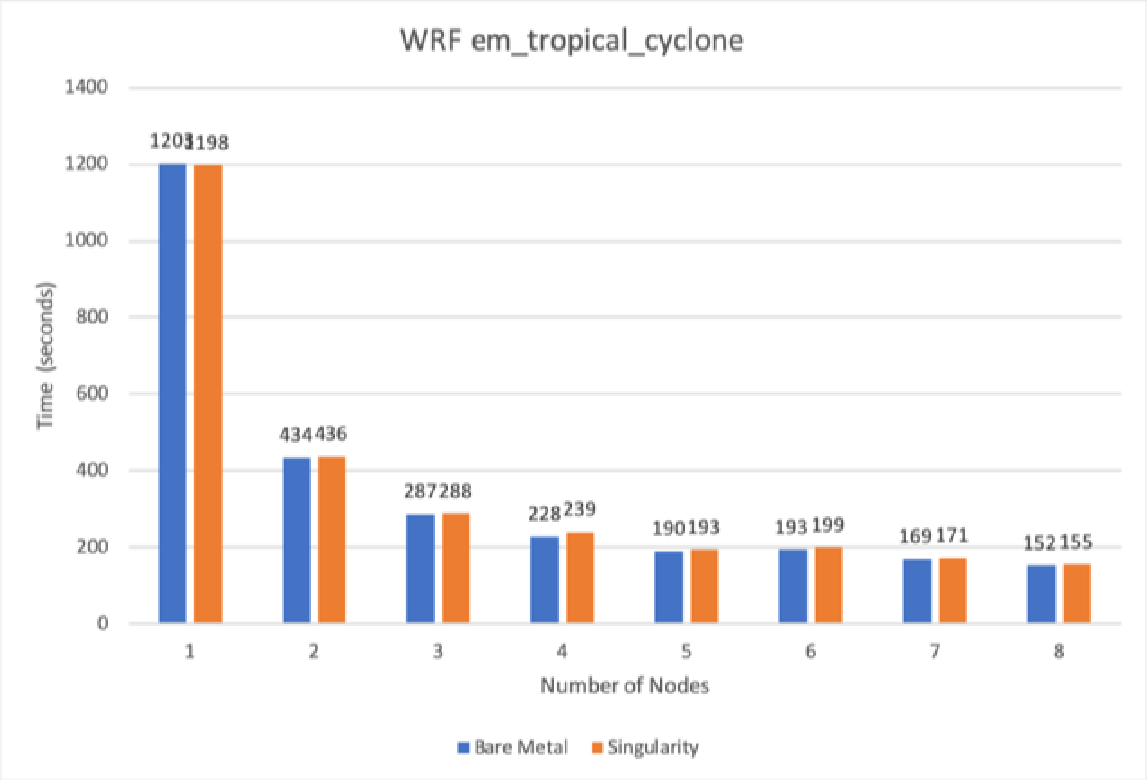


Figure 5.1: Comparison of WRF execution time on bare metal and singularity, as function of number of nodes.

6. BSC Benchmarks

6.1. Benchmarks using Lenox Cluster

We have used the Lenox cluster for comparing containerization technologies' performance. Lenox is a four computation nodes cluster where we have *sudo* capabilities, thus it is ideal to install and test different containers like Docker or Shifter, which require a more complex installation than Singularity.

Each Lenox's node has two sockets with one Intel Xeon E5-2697v3 processor. In other words, it has 14 cores per socket and 28 cores per node. In Figure 6.1, the topology of a compute node can be graphically appreciated. The network interconnecting the nodes is 1 Gigabit Ethernet over TCP. Regarding the software, it runs Linux kernel 3.10.0 with Open MPI 1.10.4 libraries. Our tested container implementations are:

- Docker 1.11.1
- Singularity 2.4.5
- Shifter 16.08.3 (BT-MZ and Alya) / 18.03.1 (HPCG)

Newer Singularity 2.X versions do not have relevant performance improvements. With Shifter, we upgraded its version to 18.03.1 with HPCG because during one maintenance of Lenox the ImageGateway of version 16.08.3 stopped working and was unable to download Docker images.

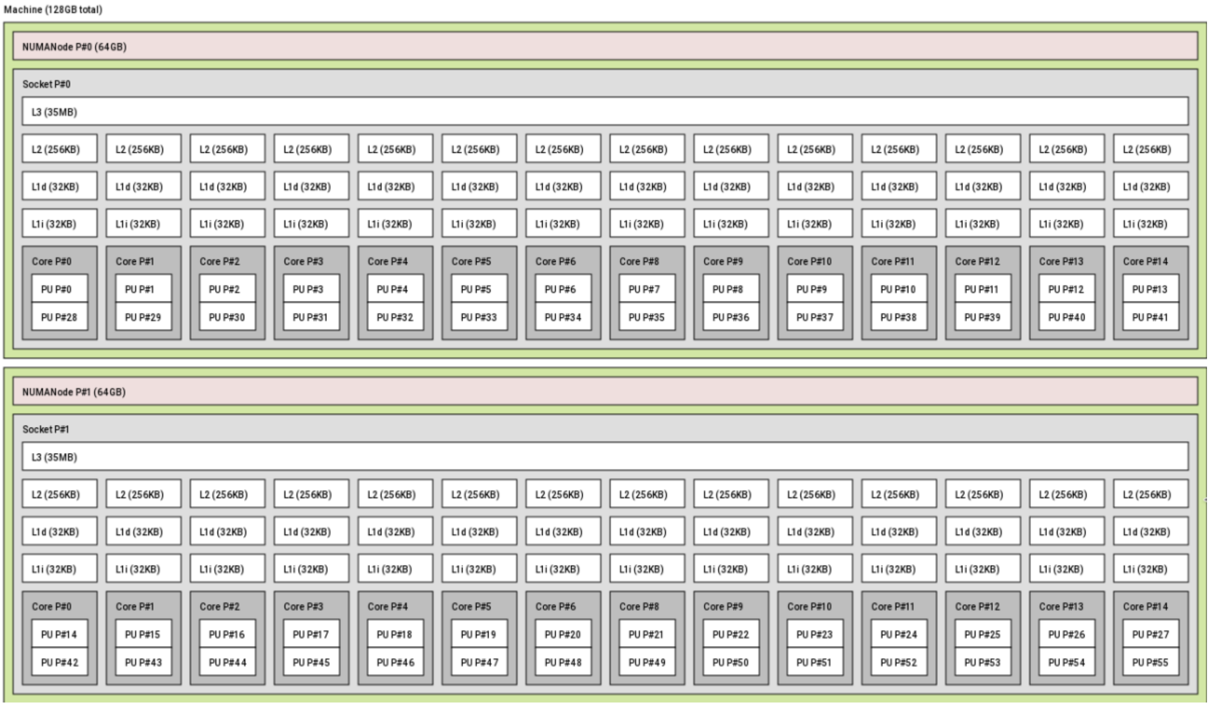


Figure 6.1: Topology of a Lenox cluster compute node

6.2. Benchmarks using MareNostrum4, CTE-POWER and ThunderX

We have used MareNostrum4, CTE-POWER and ThunderX to benchmark Alya using containers through different HPC architectures. The chosen container implementation has been Singularity because it is easy to install and secure, while Docker and Shifter present problems to be deployed in production. Table 6.1 summarizes the characteristics of each cluster. MareNostrum4, CTE-POWER and ThunderX do not have the same Singularity version because we were not responsible of its installation. Despite this, the 3 versions are very similar and should not present significant performance differences.

Table 6.1: Summary of MareNostrum4, CTE-POWER and ThunderX characteristics

	MareNostrum4	CTE-POWER	ThunderX
Number of Nodes	3.456	52	4
Sockets per Node	2	2	2
Cores per Socket	24	20	48
CPU	Intel Xeon Platinum 8160	IBM Power9 8335-GTG	Custom made Armv8-a cores within CN8890 sockets
CPU Architecture	x86	Power9	Armv8-a
Network	- Intel Omni-Path 100 Gbit/s - Ethernet 10 Gbit/s	- Mellanox EDR Infiniband 100 Gbit/s - Ethernet 10 Gbit/s	- Ethernet 40 Gbit/s
Linux Kernel	4.4.12	4.11.0	4.4.3
Singularity	2.4.2	2.5.1	2.5.2

6.3. Image building

In order to ensure that Docker, Singularity and Shifter containers have the same software stack, first we have built a common container image using Docker. Within this common image, we have all necessary compilers, MPI libraries and BSC third-party tools (i.e. programming model and performance analysis tools) to install and execute our programs. Once the base image is built, we end the image building process installing our applications with each container technology. We decided to end the building process by converting the final image from Docker to the specific container format to better leverage implementation features. We summarize the container image building process in Figure 6.2.

The final container image has the following software stack:

- Ubuntu 16.04
- Open MPI 1.10.4
- GNU 4.8.5 compilers
- BSC third-party tools
  - Nanos++ runtime [6]
  - Mercurium compiler [7]
  - DLB library [8]
  - Extrae [9]
- The application to be tested

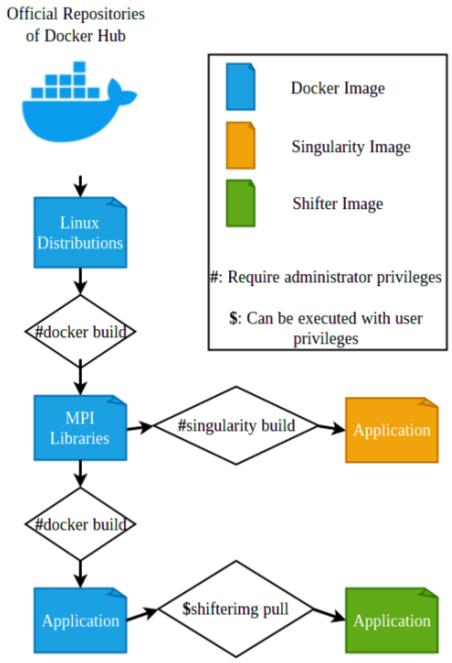


Figure 6.2: Building process of container images.

6.4. Application deployment

For our benchmarking with Alya, BT-MZ and HPCG, we have decided to perform very specific tests. In Lenox or ThunderX we cannot study scalability performance of containers, because with only 4 computation nodes our results would not represent real HPC use cases. Instead, we are going to evaluate containers' performance exploiting all of Lenox's resources (i.e. using all cores available) with different distributions of MPI ranks and threads per rank (see Table 6.2), thus evaluating the performance influence of changing these parameters.

Table 6.2: Tested distributions of ranks MPI / threads

Total number of MPI processes	MPI processes per node	Threads per MPI process
4	1	28
8	2	14
16	4	7
28	7	4
56	14	2
112	28	1

6.4.1. Bare-metal

Application’s deployment using bare-metal is straightforward, we just need to copy and compile the code avoiding the extra steps needed when using virtualization. However, since Lenox cluster does not have any workload manager as SLURM or similar, we have launched our applications manually through *mpirun* command. In essence, the complete command has the following structure:

```
$mpirun -np $N_MPIS --map-by slot:PE=$N_THREADS --bind-to core --mca
btl_tcp_if_include $NET_ADDRESS -hostfile $HOST $APP_BIN
```

Where *$N_MPIS* indicates the number of ranks MPI we want, *$N_THREADS* indicates how many threads possess each rank, *$NET_ADDRESS* specifies our network, *$HOST* is the path to our host file and finally *$APP_BIN* is our application binary execution file.

6.4.2. Docker

Application deployment with Docker is more complex, since its aim is to fully isolate containers from the host. Our first challenge consists in creating a common network for our Dockers, so they can exchange MPI messages. Once the network is operative, we must deploy a cluster of Docker containers and start the application within one container.

For inter-container communication across multiple hosts, we have used multi-host overlay networking with Etcd [10] as our external key-value store [11]. After setting up Etcd environment in each node of Lenox and creating the overlay network through Docker commands, we are able to attach our container instances to the overlay network at spawn time.

To execute the application, first it is necessary to deploy our Docker cluster. For this purpose, we launch one container instance in each node of Lenox. Thanks to the previous step (the network creation), all containers are able to communicate between them. At last, it only remains to connect to one running container and initiate the MPI execution within Docker.

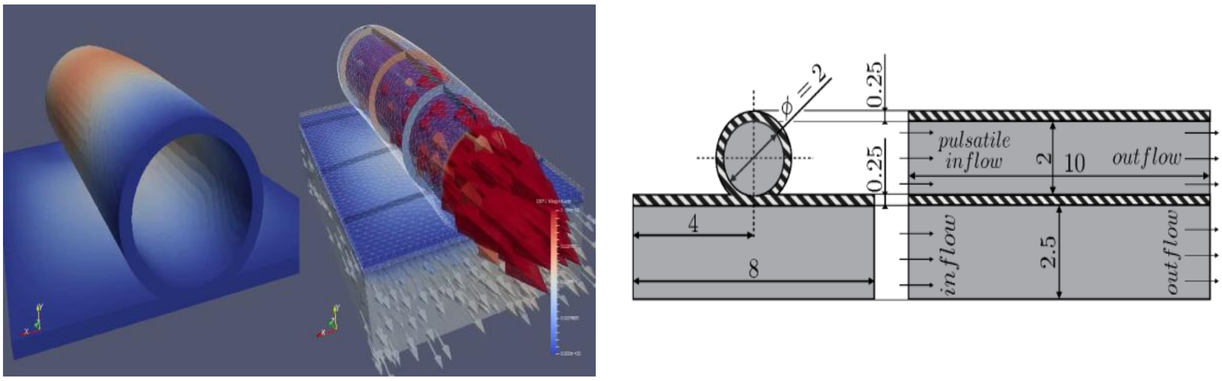
6.4.3. Singularity and Shifter

Because both Singularity and Shifter containers intend to be as integrated with the host as possible, their deployment is very simple. We only needed to execute *`mpirun`* from one node of Lenox specifying to run the appropriate container image. After invoking the MPI runtime, the host and the container implementation will manage the spawn and message communication of all application’s MPI processes.

6.5. ALYA

Alya is a large-scale in production application for multi-physics numerical simulation used at BSC to evaluate container performance with real scientific applications. Written in Fortran and parallelized using MPI and OpenMP, Alya is a highly parallel code designed to run efficiently on high-performance computers [12], [13], [14], and it is also part of the UEABS [15] (Unified European Applications Benchmark Suite), a selection of 13 scalable, portable and relevant codes for the scientific community, which made it a perfect candidate for our benchmarking purposes.

Our input for Alya consists in a use case simulating a pulsatile artery over a cavity filled with fluid (blood). This fluid-structure interaction (FSI) is depicted in Figure 6.3. The physical properties of the fluid are obtained via Computational Fluid Dynamics (CFD) and an Arbitrary Lagrangian-Eulerian (ALE) scheme, while Computational Solid Mechanics (CSM) are used to model the solid. Figure 6.3 (left) shows what this simulation looks like in a time step, and Figure 6.3 (right) is the scheme of the problem we want to solve.



a) Depiction of the simulated use case. In the left side is drawn the solid (artery) and in the right the fluid (blood).

b) Geometry used for the problem. The left side shows the short axis and the right side a section of the long axis.

Figure 6.3: Representations of our Alya use case.

Algorithm 1 shows the solving strategy used inside Alya. To solve the FSI problem, CSM(fα) represents the solver for the solid mechanic subdomain and ALE + CFD(dα) the solver for the fluid dynamics in an ALE mesh. The fluid and solid parts are simulated in a block-serial way with a Multiple Program Multiple Data strategy (MPMD), that is, each physical model is solved with its own instance of Alya. Besides, a convergence acceleration scheme $\varphi_{GS}$is used to reduce the number of coupling iterations.

```
while Time loop do
    $f_\alpha = f_\alpha^{ini}$ ;
    while Coupling loop do
        $d_\alpha = \mathrm{CSM}(f_\alpha)$ ;
        $f_\alpha = \mathrm{ALE} + \mathrm{CFD}(d_\alpha)$ ;
        $f_\alpha^{I+1} = \varphi_{GS}(f_\alpha)$ ;
    end
end
```

Algorithm 1.

From BSC’s past experience using Alya, we know that CPU and network capability are what mainly affects Alya’s performance with this use case.

6.5.1. Methodology using Alya

The Alya benchmarking is divided in two parts. First, we ran Alya in Lenox to make a comparative between different container implementations (Docker, Singularity and Shifter). Then, we executed Alya using only Singularity containers in MareNostrum4, CTE-POWER and ThunderX to study the performance obtained when using containers through different HPC architectures.

Because the FSI case requires a considerable amount of execution time to complete, and we had not available large amounts of compute resources in Lenox, CTE-POWER (only 16 nodes are available to us) and ThunderX for the kind of scaling we want to reach, we decided to run two versions of this simulation, depending on the machine capacity:

- **CFD**: simulation of the fluid domain only. Requires 2.58 times less execution time than the complete FSI simulation.
- **FSI**: simulation of the fluid-structure interaction. The real use case running the fluid and solid domains with a Multiple Program Multiple Data strategy.

We used the CFD version when benchmarking in Lenox, CTE-POWER and ThunderX, whereas the FSI has been used in MareNostrum4 supercomputer, since we had access to 256 compute nodes (12.288 cores).

Production simulations of our use case usually can be run for thousands of time steps. We broke down the duration of each time step in t_i=t_s+t_m, $\mathrm{t_i = t_s + t_m}$, measuring the two main computational phases: t_s, the duration of the *solver*, which presents a high number of MPI collectives, and t_m, the duration of the *matrix assembly*, that does not have MPI communication nor synchronization. Given the fact that all time steps are algorithmically homogeneous, we can study the average phase durations as follows:

$$\overline{t_i} = \overline{t_s} + \overline{t_m}$$

Where:

- **Average Time step duration:** $\overline{t_i} = \dfrac{\sum_{i=1}^{n} t_i}{n}$
- **Average Solver time:** $\overline{t_s} = \dfrac{\sum_{i=1}^{n} t_s}{n}$
- **Average Assembly time:** $\overline{t_m} = \dfrac{\sum_{i=1}^{n} t_m}{n}$

Our early experiments show that n = 20 is a good trade-off between overall simulation time and statistical accuracy. Figure 6.4 shows how the simulation behaves.

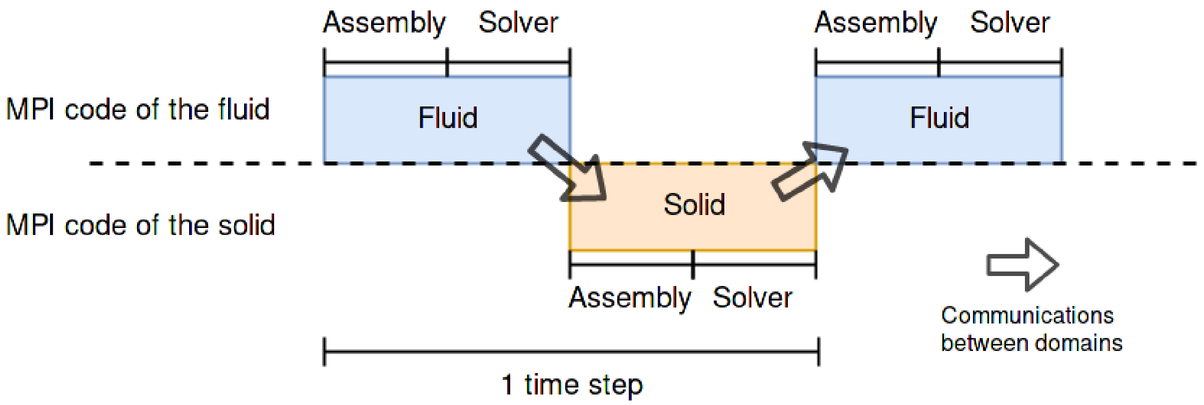


Figure 6.4: Behaviour of our Alya simulation

6.6. *BT-MZ*

BT-MZ 3.3.1 [16] belongs to the set of NAS Parallel Benchmarks, which are derived from computational fluid dynamics (CFD), focused on evaluating the parallel performance of supercomputers. From all the set of NPB, we have chosen BT-MZ (Block Tri-diagonal solver Multi-Zone) [17], [18] as it is designed to exploit multi-level parallelism and presents uneven workload allocation (i.e. load imbalance), which can be a performance killer. For these reasons, we used this benchmark to evaluate workload allocations with containers and the Dynamic Load Balancing (DLB) library in [19], [20] and now we will use it to benchmark containers. CPU, network and load imbalance are what most affect to BT-MZ's performance.

The Block-Tri-diagonal application benchmark solves a three-dimensional mesh of discretized Navier-Stokes equations. In order to act more like in production codes, NPB divides the mesh into 3-dimensional (*x*, *y* and *z*) zones across the processes. Particularly in BT, the mesh is partitioned such that zone sizes are unequal and their size grows in one direction. As a result, the application presents workload imbalance, though it is mitigated within the initialization when distributing zones among MPI ranks.

Each iteration of BT is composed by 6 functions: 1) an exchange of data with neighbour ranks, 2) computation of the right-hand side (rhs) of the matrix, solve the equations in 3) *x*-, 4) *y*- and 5) *z*-directions and 6) accumulate the results. Loops are parallelized with the OpenMP pragma *PARALLEL DO* [21]. Data between MPI ranks is exchanged with non-blocking sends and receives while processes are synchronized with waitalls.

```
do step 1, niter
     call exch_qbc
     do zone 1, num_zones
          call compute_rhs
          call x_solve
          call y_solve
          call z_solve
          call add
     end do
end do
```

Algorithm 2. BT-MZ's code structure.

6.6.1. Methodology BT-MZ

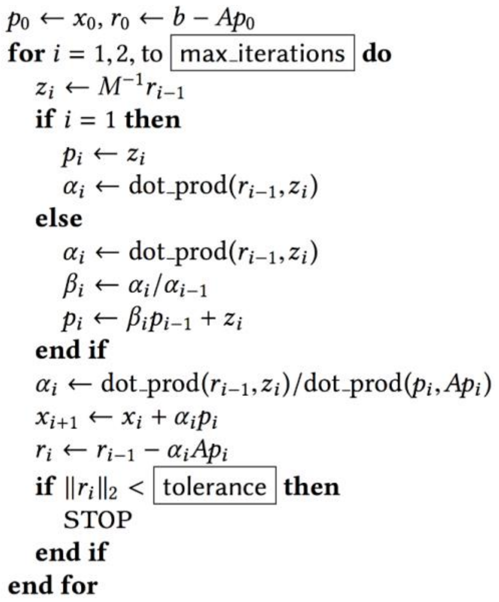
We have compiled NPB BT-MZ 3.3.1 benchmark with gfortran 4.8.5 because it was the latest version of the Fortran compiler available in Lenox. We have executed the hybrid MPI+OpenMP version of BT-MZ (to benchmark both programming models) using the class C problem size with the default iteration configuration. BT-MZ offers different problem sizes, from the S size (the smaller) to the F (the bigger). The C problem size is the medium one, which allowed us to repeat the executions without spending much time. After each execution of BT-MZ, we have collected its execution time and total Mop/s reported by the benchmark.

BT-MZ has been executed 50 times, which is a good choice between statistical accuracy and overall execution time, in Lenox with each technology: bare-metal, Singularity, Shifter and Docker. Each technology has used Open MPI 1.10.4 libraries (the latest version available in Lenox) and TCP over Ethernet network to pass inter-node MPI messages.

6.7. HPCG

The (HPCG) benchmark [22], [23] is intended to emulate scientific applications which solve Partial Differential Equations (PDEs). HPCG implementation is very similar to the well-known High-Performance LINPACK benchmark, except that HPCG focuses on stressing the memory design. HPCG benchmark tries to cover the most common communication and computation patterns emphasizing on high-performance collectives and local memory design.

HPCG performs a symmetric Gauss-Seidel preconditioned conjugate gradient solver on a sparse linear system. In the setup phase, HPCG generates a synthetic discretized three-dimensional equation model. This three-dimensional matrix will be decomposed with a conjugate gradient method for the resulting sparse linear system. Afterward, HPCG performs a loop where it operates with the sparse system as represented in Algorithm 3.



```
$p_0 \leftarrow x_0, r_0 \leftarrow b - Ap_0$
for $i = 1, 2,$ to $\boxed{\text{max\_iterations}}$ do
  $z_i \leftarrow M^{-1} r_{i-1}$
  if $i = 1$ then
    $p_i \leftarrow z_i$
    $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)$
  else
    $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)$
    $\beta_i \leftarrow \alpha_i / \alpha_{i-1}$
    $p_i \leftarrow \beta_i p_{i-1} + z_i$
  end if
  $\alpha_i \leftarrow \text{dot\_prod}(r_{i-1}, z_i)/\text{dot\_prod}(p_i, Ap_i)$
  $x_{i+1} \leftarrow x_i + \alpha_i p_i$
  $r_i \leftarrow r_{i-1} - \alpha_i A p_i$
  if $\|r_i\|_2 < \boxed{\text{tolerance}}$ then
    STOP
  end if
end for
```

Algorithm 3

HPCG is a valuable and representative benchmark because of its adoption in the Top 500. As HPCG is designed to measure the performance of basic operations in supercomputing and to stress the host, we believe that its benchmarking is both valuable and authentic for possible use cases.

6.7.1. Methodology HPCG

HPCG has been compiled using g++ 4.8.5 compiler with -O3 optimizations and Open MPI 1.10.4 libraries (the latest version of the g++ compiler available in Lenox and the default optimization flags of the configure). In addition, HPCG is compiled with both MPI and OpenMP features enabled. At last, we executed 25 times HPCG in Lenox with each technology (bare-metal, Singularity, Shifter and Docker) specifying local subgrid dimensions of 120 for x, y and z-axis because it offered a good trade-off between the number of repetitions and elapsed time of the execution. After each execution we collected the log file generated with its corresponding metrics.

6.8. Comparison between containers

In this Section, we compared Docker, Singularity and Shifter container implementations in the Lenox cluster, in terms of performance having as a reference bare-metal executions. With each technology, we tested various MPI+OpenMP distributions to check their influence in performance, by collecting at the end of the execution the metrics reported by the application. Our containers used the same MPI libraries versions as the host, which are Open MPI 1.10.4.

6.8.1. Alya

In Figure 6.5, we display the average step duration of Alya with every technology (bare-metal, Singularity, Shifter and Docker) using all 112 cores of Lenox. In the *x*-axis we represent the different MPI+OpenMP distributions, and in the *y*-axis the time in seconds with its standard deviation. Here, HPC designed containers (i.e. Shifter and Singularity) are able to match bare-metal performances maintaining similar variations in any distribution of processes. In the case of Docker, in 8x12 (i.e. 8 MPI ranks and 12 OpenMP threads) and 16x7 distributions, its performance is equal to the other technologies, but from 28x4 onwards, it experiences some sort of overhead which is most notable in 112x1.

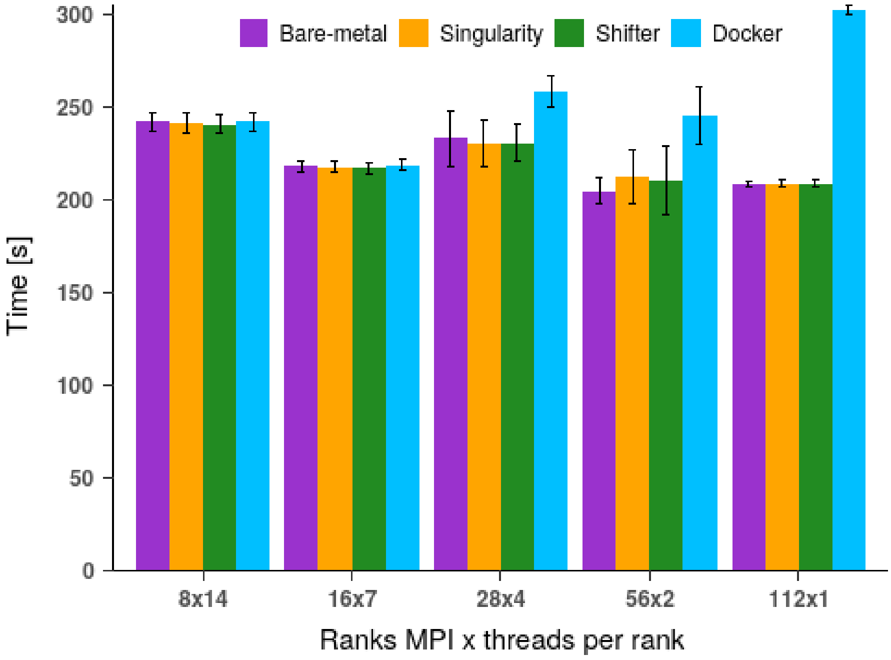


Figure 6.5: Average step duration time of Alya in Lenox.

We are able to better understand this performance difference analysing the two phases of the simulation separately. In Figure 6.6, we show the elapsed time of the matrix assembly phase and in Figure 6.7 the elapsed time of the solver phase. The axes of both charts display the same information as in Figure 6.5.

As we explained before, the assembly phase is computer intensive and does not contain MPI communication. Therefore, Docker, Singularity and Shifter containers do not add any significant overhead to the computational performance, and what is more, the performance obtained is identical matching almost perfectly their means and variations in all our process distributions. On the other hand, inspecting the solver time in Figure 6.7, it is clear that the main source of performance loss or variation when using containers is communication. The solver phase contains a high amount of MPI collective communications, being this the reason for the performance loss of the Docker container since its containers use here an extra network layer (the virtual network we deployed).

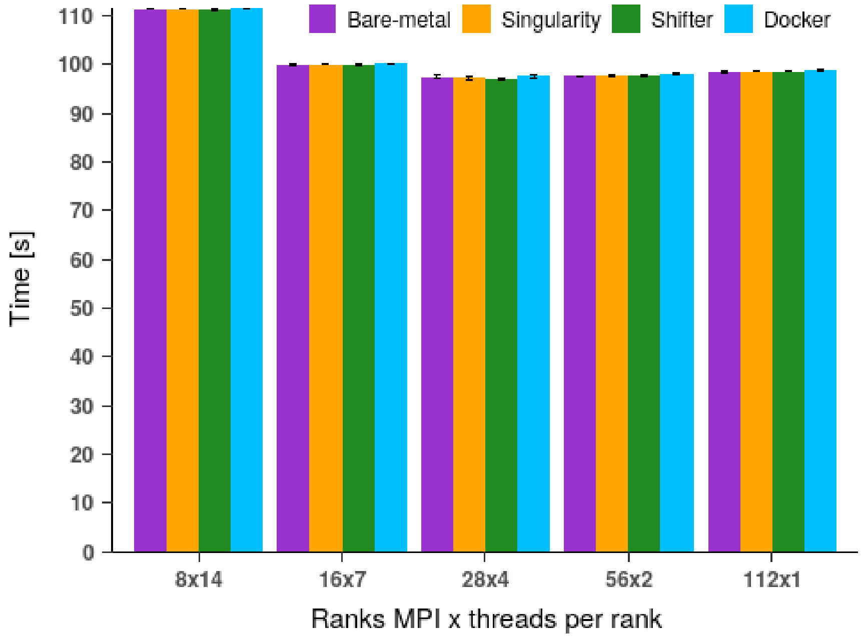


Figure 6.6: Average assembly time of Alya in Lenox.

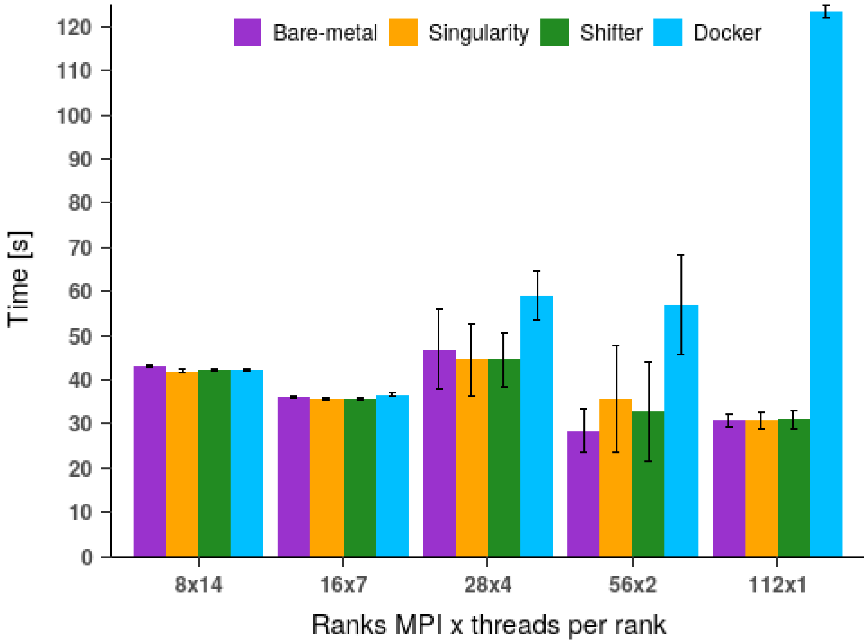
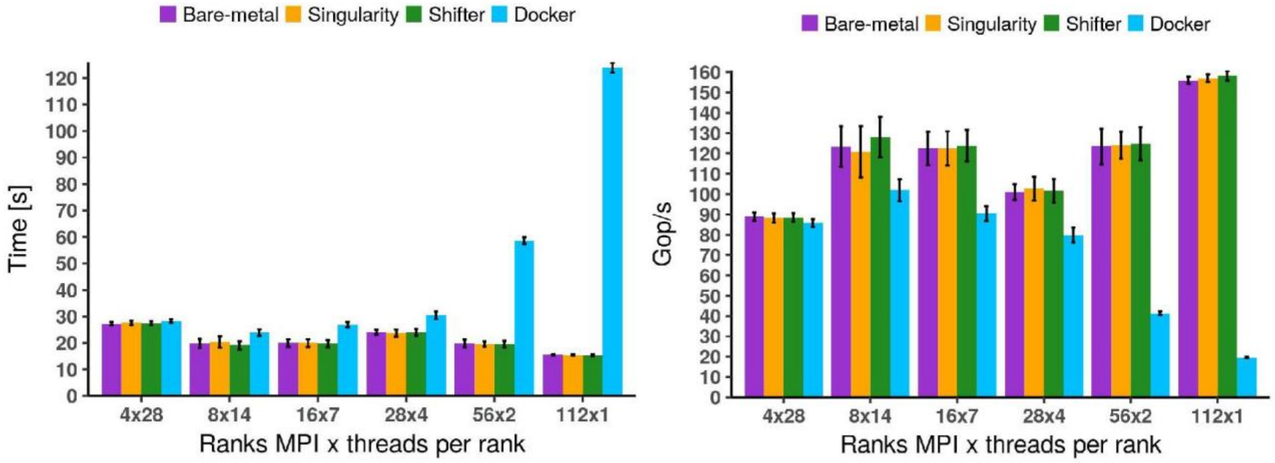


Figure 6.7: Average solver time of Alya in Lenox.

6.8.2. BT-MZ

Figures 6.8a and 6.8b show the performance obtained from BT-MZ. In both charts, the *x*-axis represents the MPI+OpenMP distribution tested, whereas the *y*-axis of Figure 6.8a displays the average execution time (less is better) and the *y*-axis of Figure 6.8b the average Gop/s obtained (10^9operations per second, more is better) with their standard deviation.



a) Average Execution time.

b) Average operations per second.

Figure 6.8: BT-MZ performance in Lenox.

Thanks to these figures, we can appreciate the performance difference between our 4 execution environments (bare-metal, Singularity, Shifter and Docker). The execution time difference reported by Figure 6.8a between bare-metal, Singularity and Shifter with all distributions is minimal. In Figure 6.8b, we can see again how bare-metal and the HPC designed containers (Singularity and Shifter) reach almost identical performances in operations per second. Finally, it is evident how Docker's performance gets worse as we scale in the number of MPI processes: in 4x28 its performance matches the other technologies, but from there, its execution time

This performance degradation is due to the MPI communications as we could check when enabling the detailed timers of BT-MZ. In Table 6.3, we expose the profiling of one BT-MZ execution with each distribution and technology. Table 6.3a shows the time in seconds that BT-MZ spends in CPU and MPI communications phases, whereas Table 6.3b shows the same as a percentage of the total execution time. As we can notice in Table 6.3a, the time BT-MZ spends with Docker in communications tends to duplicate linearly with the duplication of the number of processes. For instance, from 28x4 to 56x2 the communication time increases 28,73 seconds, and from 56x2 to 112x1 it grows again 65,09 seconds. This increase, however, does not happen in the CPU time, which maintains similar to the other technologies. On the contrary, the obtained metrics with bare-metal, Singularity and Shifter are alike and do not present such a drastic increase in the communication phase. Given the fact that the only difference between Docker and the other technologies is the way it is deployed, we assume that this overhead is caused by the virtual network Docker is using here.

Table 6.3: BT-MZ profiling with different distributions and technologies

	Bare-metal		Singularity		Shifter		Docker	
Phase Distribution	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm
4x28	14.73	12.81	15.06	12.49	14.68	12.78	15.00	13.31
8x14	8.26	11.61	8.36	12.04	8.23	10.88	8.25	15.65
16x7	7.06	12.88	7.06	12.89	7.08	12.64	7.03	19.84
28x4	6.63	17.43	6.58	17.13	6.61	17.37	6.60	23.86
56x2	6.08	13.72	6.97	13.55	6.09	13.45	6.09	52.59
112x1	5.69	9.86	5.70	9.75	5.70	9.63	6.44	117.68

a) Time (seconds) that BT-MZ spends in CPU and MPI communications

	Bare-metal		Singularity		Shifter		Docker	
Phase Distribution	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm	CPU	MPI comm
4x28	54.10	46.90	54.70	45.30	53.50	46.50	53.00	47.00
8x14	41.70	58.30	41.20	58.80	43.20	56.80	34.60	65.40
16x7	35.50	64.50	35.50	64.50	36.00	64.00	26.20	73.80
28x4	27.60	72.40	27.80	72.20	27.60	72.40	21.70	78.30
56x2	30.80	69.20	31.00	69.00	31.20	68.80	10.40	89.60
112x1	36.60	63.40	36.90	63.10	37.20	62.80	5.20	94.80

b) Percentage that BT-MZ spends in CPU and MPI communications

6.8.3. HPCG

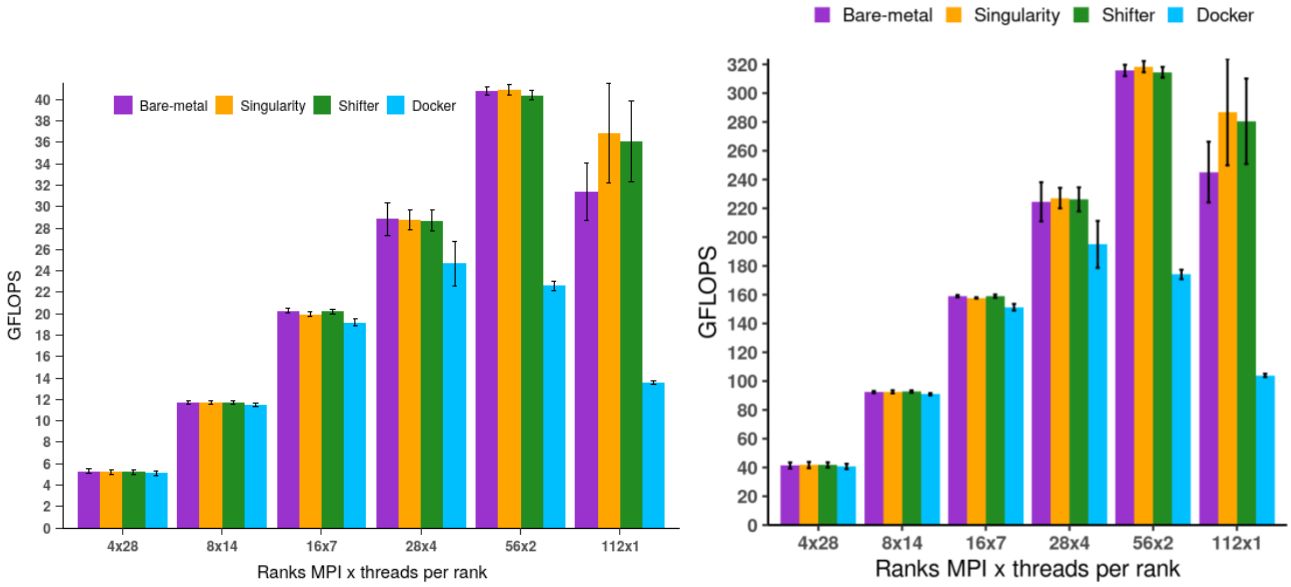
Figure 6.9a displays the average GFLOPS[1] obtained by HPCG in Lenox. As with previous benchmarks, in the *x*-axis we are testing various MPI+OpenMP distributions, and the *y*-axis shows the average GFLOPS with its standard deviation. These results follow the same pattern as with BT-MZ, since Singularity and Shifter match bare-metal performances, while Docker gets worse GFLOPS as we increase the number of processes. In these executions, the 112x1 distribution presents two anomalies: the bare-metal performance is significantly worse than Singularity or Shifter (approximately 5 GFLOPS, 16,6% of relative difference with respect to Singularity); and the standard deviation in this distribution is very large except with Docker.

Inspecting more this specific case (112x1), we have seen that the cause of this performance difference between bare-metal and containers is the bandwidth. In Figure 6.9b we show the average bandwidth in GB/s each technology gets. The bar plot appears to be identical of Figure 6.10a, nevertheless, this bandwidth reported by the application is the sum of the network and memory, so it is not clear which component misbehaves.

With further analysis of the metrics reported, we found that only one function (*ComputeDotProduct_ref*) of HPCG presents different execution times in bare-metal and within containers. This function performs the scalar product of two vectors and has involved one *MPI_Allreduce()*, an MPI collective communication function. For this reason, we believe that the performance difference of 112x1 is due to network communications through MPI. Anyway, we were still unable to find the ultimate cause of this network bandwidth difference in the bare-metal case, because the Lenox cluster was decommissioned during the final stages of benchmarking. Though, we suppose that this might be due to the difference in the compiler libraries available in the bare-metal

[1] Thousands of millions (10^9) floating point operations per second.

system (Red Hat Enterprise Linux 7 distribution) and the containerized environment (Ubuntu 16.06 distribution).



a) Average floating point operations.

b) Average memory and network bandwidth.

Figure 6.9: Performance of HPCG in Lenox.

6.9. Comparative Conclusions

We have tested Docker, Singularity and Shifter with 3 different applications, one of them a real HPC multiphysics simulator, Alya. Every benchmarking has showed that the HPC designed containers (Singularity and Shifter) can match either the CPU, network or memory bare-metal performances. In addition, these containers also have proved that their performance compared with bare-metal does not get worse when deploying them with different MPI+OpenMP configurations. On the contrary, though Docker containers do not add overhead to the CPU nor memory usage, their network performance seems to be very bad when scaling in the number of processes due to its deployment complexity and level of isolation. In regards on how to solve this Docker issue, it could be possible to explore other alternatives when deploying large MPI applications using container orchestration software (like Kubernetes) or searching other ways to interconnect Docker containers.

The containers comparative with Alya, along with more results (deployment overhead comparison) [24] was presented at the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).

6.9.1. Portability and Scalability Evaluation of Containers

Using Singularity, we executed Alya through 3 different state-of-the art HPC systems (MareNostrum4, CTE-POWER and ThunderX) and studied its performance. Unlike in the comparison between containers, here we have executed the pure MPI version of Alya because we noticed that the number of threads each process has does not alter the performance.

Having in consideration that MareNostrum4 and CTE-POWER have high-performance MPI networks, we decided to test two different deployment strategies with Singularity.

- **Generic container**: we built a container image with a custom set of libraries installation (e.g. MPI). This container is easier to build since the user ignores the characteristics of hosts, but in exchange this image is unable to leverage hardware specific components like GPUs, high-performance networks, etc.
- **Integrated container**: we built a container image which is able to load at runtime the host's MPI libraries. This image brings more work to be built as the user needs to identify the host MPI libraries and prepare the container to load them, as can be seen in [25]. However, this container leverages the optimal hardware of its host.

The integrated container is made to leverage the Intel Omni-Path and Mellanox Infiniband networks of MareNostrum4 and CTE-POWER respectively, whereas the generic container is limited to communicate through the generic Ethernet network of its corresponding hosts.

MareNostrum4

In MareNostrum4 we had available up to 256 computational nodes, so we decided to perform a strong scalability test running the FSI case of Alya. We show the results obtained in Figure 6.10 where the *x*-axis displays the number of nodes we used in each execution, and the *y*-axis the speedup each version of Alya obtained with respect to its initial execution with 4 nodes. We can see clearly how the integrated container reaches bare-metal speedups with any number of resources, while the generic container only can do this up to 32 nodes. With more than 32 nodes, the generic container diverges from the ideal speedup, losing performance starting from 64 nodes. Actually, we were unable to run the generic container with 256 nodes because some of its processes could not establish connection with their MPI neighbours, freezing the execution of the application inside when it reaches an MPI barrier. Since the generic container is using the TCP protocol for MPI communications, which implements a connection time out, the more we saturate the network increasing the amount of MPI processes, the more prone are their sockets to suffer this error. Since we do not have the rights to increase the TCP "connection time out" time in MareNostrum4, we could only repeat our executions hoping to avoid this issue, but with more than 200 nodes it was too unlikely to happen.

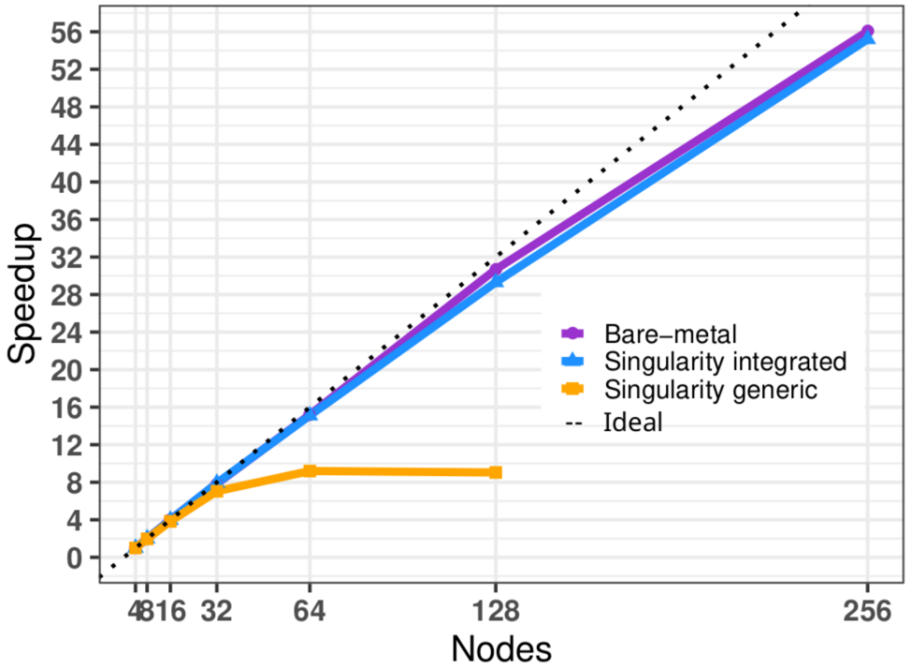


Figure 6.10: Alya FSI strong scalability test in MareNostrum4

CTE-POWER

In Figure 6.11 we display the average elapsed time of each container version of Alya. The *x*-axis shows the number of nodes we have used and the *y*-axis the time in seconds. We can appreciate how the integrated container gets close-to bare-metal execution times in all the chart, whereas the generic container experiences some sort of overhead which increases as we use more nodes.

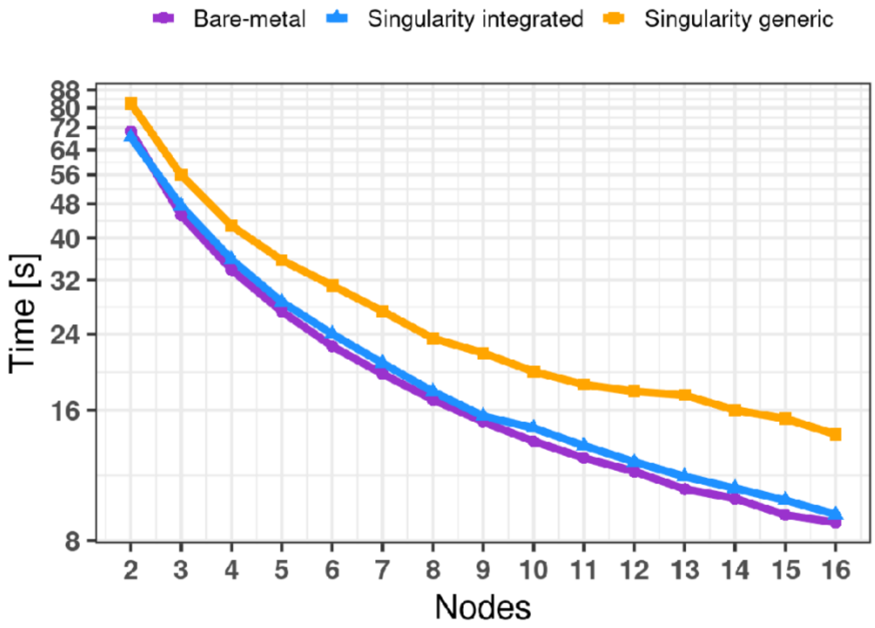
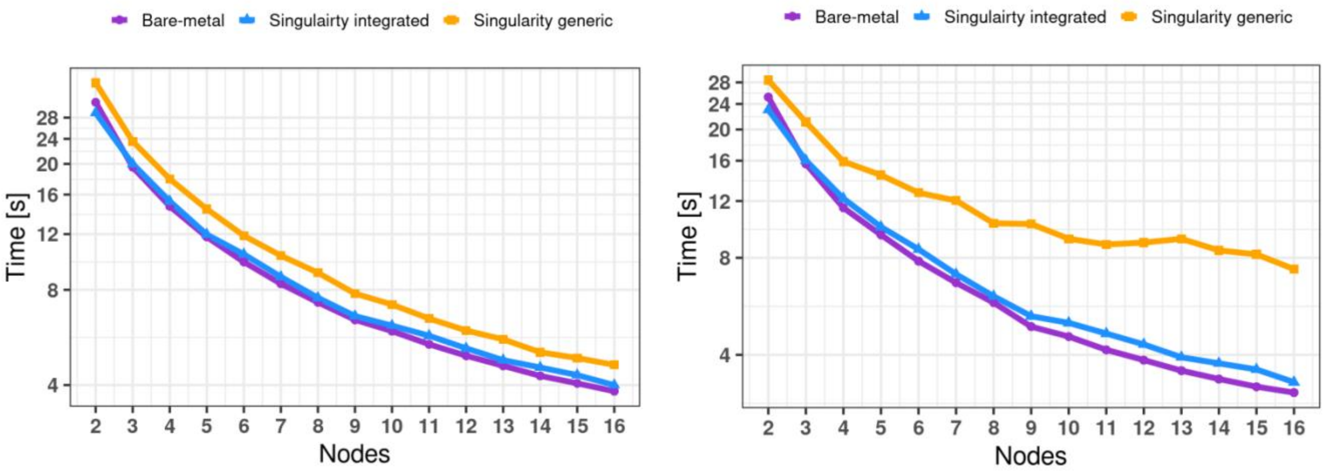


Figure 6.11: Average elapsed time of Alya CFD in CTE-POWER.

To better understand why the generic container performs better we display Figure 6.12, where Figure 6.12a shows the average time of the assembly phase and Figure 6.12b of the solver phase. In the assembly phase, the time difference between bare-metal and the generic container is noticeable, but remains constant with any number of nodes and is not significantly worse. It is in the solver phase where most of the overhead appears, because increasing the number of nodes also increases the solver phase time of the generic container. This fact is obvious, since the generic container is unable to use the Mellanox EDR network of CTE-POWER, and therefore, as we increase the number of processes, we are congesting the Ethernet network, which is not prepared for such workloads.



a) Average assembly time.

b) Average solver time.

Figure 6.12: Alya CFD performance of its phases in CTE-POWER.

ThunderX

In Figure 6.13, the average time step from the executions of Alya CFD in ThunderX are shown. The results we got here are specially interesting because in this case all three versions of Alya get the exact same elapsed time, even the generic container, with any number of nodes. The reason for this is that ThunderX does not possess any high-performance network, or in other words, it does not have specialized hardware that our containers need to leverage explicitly. Therefore, since all three versions are executing Alya with the same hardware resources, they get the same results.

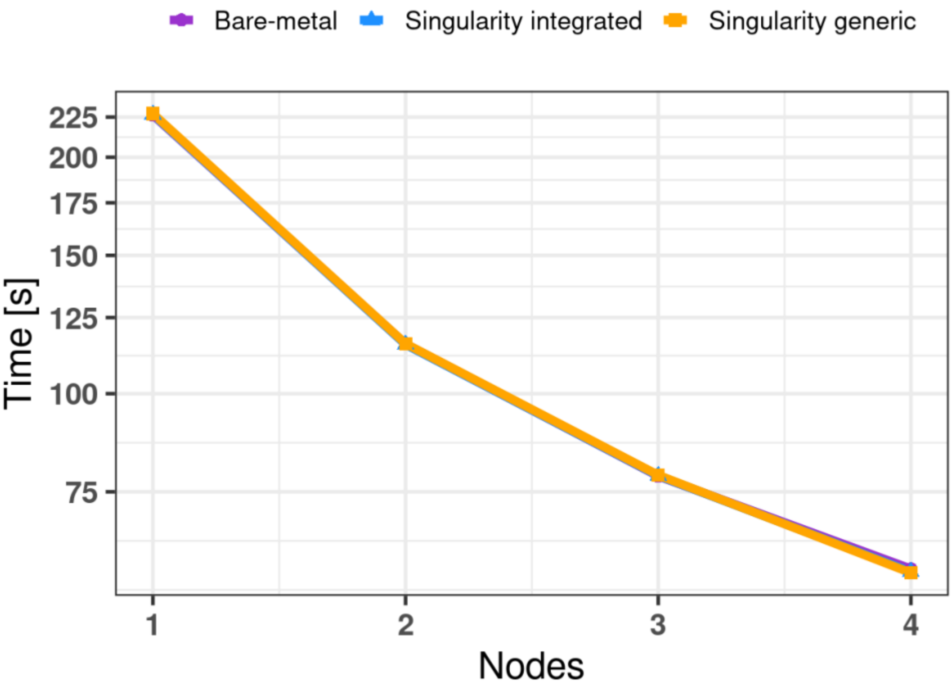


Figure 6.13: Alya CFD average time step in ThunderX.

6.9.2. Portability and Scalability Conclusions

We have performed a large scalability test in MareNostrum4 using a real HPC use case (Alya) and tested containers' performance with two state-of-the art HPC architectures (IBM Power9 and Armv-8a). Because some of our testbeds possessed specialised hardware, i.e. the high-performance networks Intel Omni-Path of MareNostrum4 and Mellanox EDR Infiniband of CTE-POWER, we have deployed two types of containers: the integrated one, capable of leveraging its host specific hardware; and the generic one, which is only prepared to use the by default hardware components.

We have demonstrated that containers can reach bare-metal performance in any case, with small or very large number of resources. Even the generic container can do that if the host does not present hardware that requires further configuration. Containers do not add CPU, memory or network overhead, at least when accessing the hardware. It might occur that executing applications with containers make them perform worse, but actually this is more related with the software within the built container (the libraries, whether the applications are configured with optimization flags, drivers to use high-performance networks or GPUs, ...) rather than the virtualization overhead, which in HPC designed containers like Singularity tends to be zero.

This portability and scalability study was presented at the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS) [24].

7. CINECA Benchmarks

7.1. Benchmark System

The cluster used for testing containerization technologies is the Tier-0 system MARCONI [26] Lenovo NeXtScale platform co-designed by CINECA. It offers the scientific community a technologically advanced and energy-efficient high performance computing system. The partition of MARCONI used for the testing is named “A3 partition” which is a Lenovo Stark, made by 21 racks, and more than 2’300 compute nodes, each of them equipped with 2 x 24-cores Intel Xeon 8160 CPU (Skylake) at 2.10 GHz and 192 GB of RAM DDR4.

This supercomputer takes advantage of the Intel® Omni-Path Architecture, which provides the high-performance interconnectivity required to efficiently scale out the system’s thousands of servers. A high-performance Lenovo GSS storage subsystem, that integrates the IBM Spectrum Scale™ (GPFS) file system, is connected to the Intel Omni-Path Fabric and provides data storage capacity.

7.2. Image building

Following the usage scheme proposed in D12.1 [27], the Singularity image files have been built on an external server and then copied on MARCONI cluster.

A "built from scratch" approach has been followed. In some cases, it has been chosen to install all the needed software at the same time using a single recipe file. Other times, a "Matryoshka" approach has been chosen, in which a singularity container has been built starting from another one previously built and then adding other software.

Two codes have been tested, Quantum Espresso [28] and Tensorflow [29].

7.3.TENSORFLOW

Tensorflow [29] is a widely used open source machine learning library for research and production. It is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in Machine Learning and developers easily build and deploy Machine Learning powered applications.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence Research organization for the purposes of conducting machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

7.3.1. Results

According to the philosophy of reproducibility and sharing that characterize the container utilization, it has been decided to use the pre-built, official released container. The containers built have been bootstrapped from the Docker ones available in the official Docker Hub repository of Tensorflow community [30].

The version of Tensorflow used is the 1.10.0 for CPU.

A singularity container was built, bootstrapping from that available in Docker hub and adding the HOME path directory of the cluster used and a test directory that was used to bind the directory with the results physically located on the cluster. So, the recipe used to build the container was:

```
*****************************************************
Bootstrap:docker
From:tensorflow/tensorflow:1.10.0
%post
mkdir -p /test
chmod 777 -R /test

mkdir -p /marconi
chmod 777 -R /marconi
*****************************************************
```

The container has been built using Singularity 3.0.2 version, the same also available as a module on MARCONI cluster.

Both for bare metal and container tests, five different neural networks have been considered: AlexNet [31], googLeNet [32], InceptionV3 [33], ResNet-50 [34] and VGG16 [35]. The dataset was ImageNet [36] (synthetic) and two different batch sizes were analysed: 32 and 64. The tests have been repeated 3 times and averaged on the set of data. All the runs have been executed in a single node of MARCONI Sky Lake.

The number of images per second is reported in the Figures 7.1 and 7.2. The batch size being fixed, each histogram shows the number of images per second computed in each neural network model described above.

As shown, important overhead in the execution of containerized run is introduced using the container instead of the bare metal application, because tensorflow inside container is not optimized for the architecture used, for portability reasons. In fact, on the output of the containerized run is written.

```
"[...] tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU
supports instructions that this TensorFlow binary was not compiled to
use: AVX2 AVX512F FMA"
```

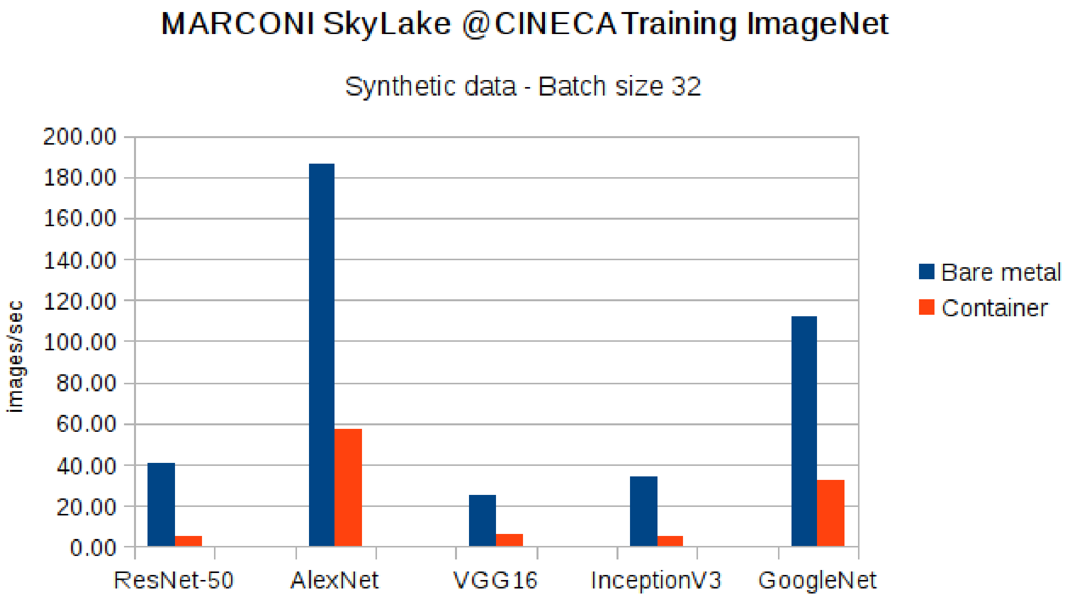


Figure 7.1: Number of images per second computed in ResNet-50, AlexNet, VGG16, InceptionV3 and GoogleNet model on a single MARCONI Sky Lake node. The batch size used is 32, the dataset is ImageNet - synthetic

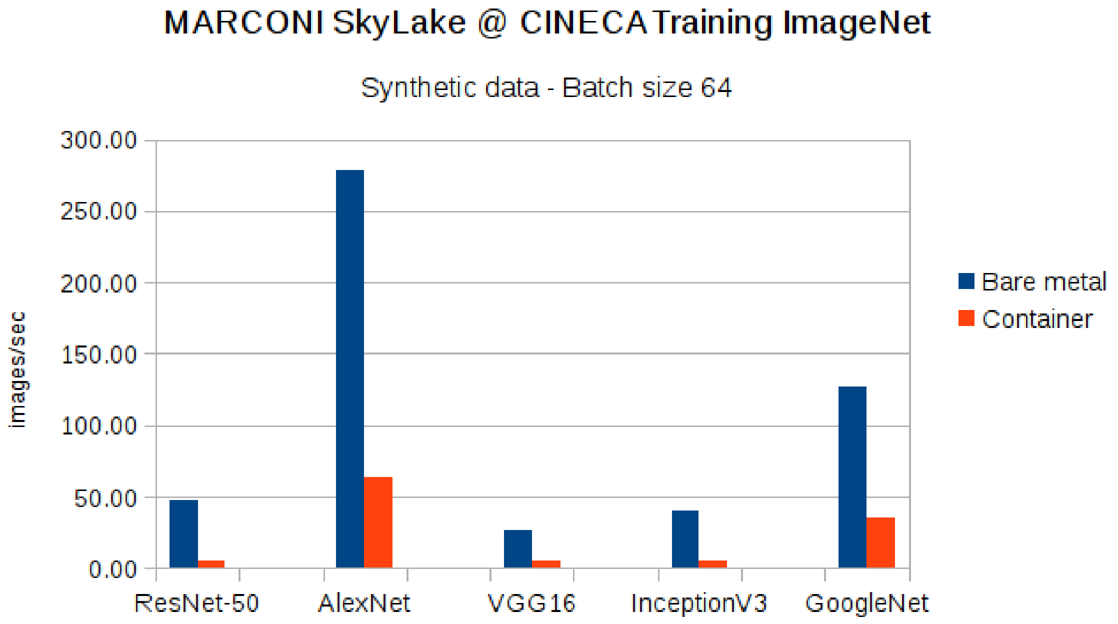


Figure 7.2: Number of images per second computed in ResNet-50, AlexNet, VGG16, InceptionV3 and GoogleNet model on a single MARCONI Sky Lake node. The batch size used is 64, the dataset is ImageNet - synthetic

7.4. QUANTUM ESPRESSO

Quantum Espresso [28] (QE) is an integrated suite of Open-Source computer codes for electronic-structure calculations and materials modelling at the nanoscale. It is based on density-functional theory, plane waves, and pseudopotentials.

As manifest in the official page, Quantum Espresso has evolved into a distribution of independent and inter-operable codes in the spirit of an open-source project. The QE distribution consists of a "historical" core set of components, and a set of plug-ins that perform more advanced tasks, plus a number of third-party packages designed to be interoperable with the core components. Researchers active in the field of electronic-structure calculations are encouraged to participate in the project by contributing their own codes or by implementing their own ideas into existing codes.

QE is an open initiative, in collaboration with many groups world-wide, coordinated by the QE Foundation. Present members of the latter include Scuola Internazionale Superiore di Studi Avanzati (SISSA), the Abdus Salam International Centre for Theoretical Physics (ICTP), the CINECA National Supercomputing Center, the Ecole Polytechnique Fédérale de Lausanne, the University of North Texas, the Oxford University. Courses on modern electronic-structure theory with hands-on tutorials on the QE codes are offered on a regular basis in collaboration with ICTP.

The following calculations can be done using QE:

- Ground-state calculations
- Structural Optimization, molecular dynamics, potential energy surfaces
- Electrochemistry and special boundary conditions
- Response properties (DFPT)
- Spectroscopic properties
- Quantum Transport

The application runs on almost every conceivable current architecture: from large parallel machines (IBM SP and BlueGene, Cray XT, Altix, Nec SX) to workstations (HP, IBM, SUN, Intel, AMD) and single PCs running Linux, Windows, Mac OS-X, including clusters of 32-bit or 64-bit Intel or AMD processors with various connectivity (gigabit ethernet, myrinet, infiniband…). It fully exploits math libraries such as MKL for Intel CPUs, ACML for AMD CPUs, ESSL for IBM machines.

QE software can be downloaded at [37]. A GPU-enabled version is also available.

7.4.1. Results

In the presented tests, pure mpi version of QE has been used. At the beginning of a QE simulation, the real space domain is distributed among the mpi tasks using collective mpi communications. After such subdivision, each task executes electronic structure calculations with linear algebra routines. Then the date are gathered to compute the total energy of the system. If it is over the chosen energy threshold, the data are again split among processors and new computation are done. This cycle is repeated until the total energy convergence. So, in QE there are both intensive communications among tasks and intensive linear algebra calculations.

Our tests were focused mainly on a performance comparison among bare-metal and container execution on MARCONI using different compiler and network drivers. So, two containers have been built and used for test: one that uses licensed software (Intel MPI compiler) and takes care of the fact that on MARCONI the Intel OmniPath network is available among compute nodes. The other, instead, is totally free of licenced software, and a generic network driver has been installed.

Finally, in both the two containers, all the needed environment variables have been set in the "**`%environment`**" section of the Singularity recipe, in order to be available at run time.

Three tests have been done with such containers. The result for Test case 1 and Test case 2, described in the next section and already showed in Deliverable 12.3 of HPC-Europe3 project [38], are here summarized. Moreover, some comments about QE code structure and about the mineral used are provided. The third test executed was focused in scalability performance by increasing the number of nodes used during the simulation. The tests have been accepted at the "*1st International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE HPC)*".

Test case 1

The version of QE used is 6.3, compiled with Intel parallel studio 2018 - update 4. The container has been built using Singularity 3.0.1.
In the containerized version, the Intel OmniPath driver has also been installed, with the same version that on MARCONI SkyLake where the tests have been made, i.e. 10.5.0.0-140.
For the tests, we used a small and a larger slab of the same material: the Zirconium Silicide [38]. Such structure has been extensively studied within QE official benchmarks [39] because it has a periodic lattice and it easily converges at its total energy ground state. Moreover, since it is made by atoms with many electrons, to simulate such material, hundreds GBs of RAM are needed, that can be found in a HPC cluster.
We used two different slab of the same structure, to leverage the scalability of the RAM used during the computation. The number of atoms chosen in such slabs depends on the symmetry of the mineral. More precisely the dimension of the slabs is:

- small: 24 atoms in total with a K-point mesh of 6 x 19 x 13
- big: 108 atoms in total with a K-point mesh of 6 x 6 x 5

Pure MPI Quantum Espresso simulations are reported in the two graphs below, Fig. 7.3 and 7.4 up to 10 Sky Lake nodes, i.e. up to 480 MPI tasks. The comparison between the application performance of bare metal vs container for both the above inputs is shown. Total execution time is reported for the entire code (PWSCF) and for the Fastest Fourier Transform in the West (fftw), that is one of the most time-consuming parts of the code. The calculations were repeated 5 times and then averaged on the set of data. In these simulations, the Intel OmniPath interconnection networks among the compute nodes has been used. As it can be noted, the use of container doesn't introduce significant overhead, since the total execution time of bare metal and container runs are about the same, up to 480 cores (10 Sky Lake compute nodes).

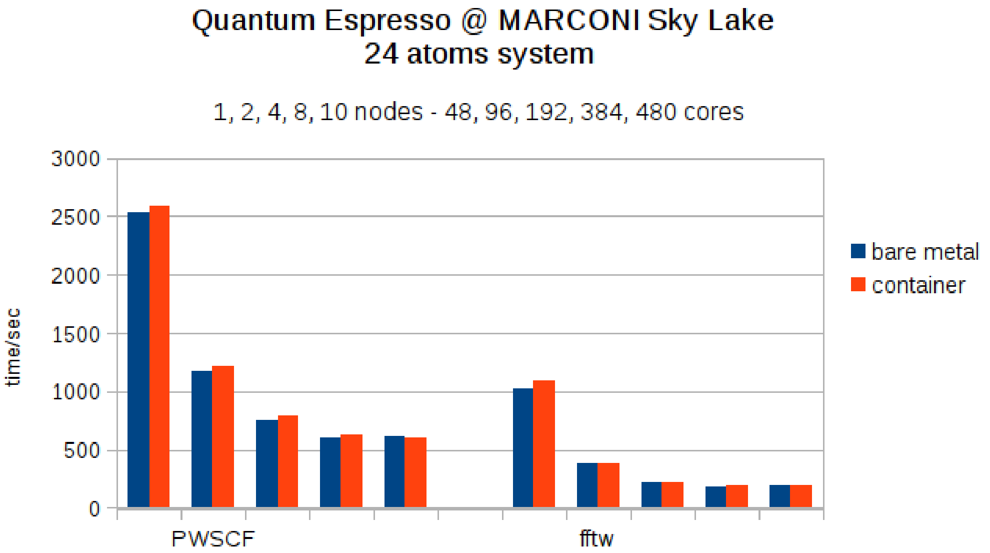


Figure 7.3: Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (fftw) are graphed for a system of 24 zirconium and silicon atoms run up to 10 MARCONI Sky Lake nodes. Both in the container and bare-metal execution, QE code is compiled using Intel MPI compiler.

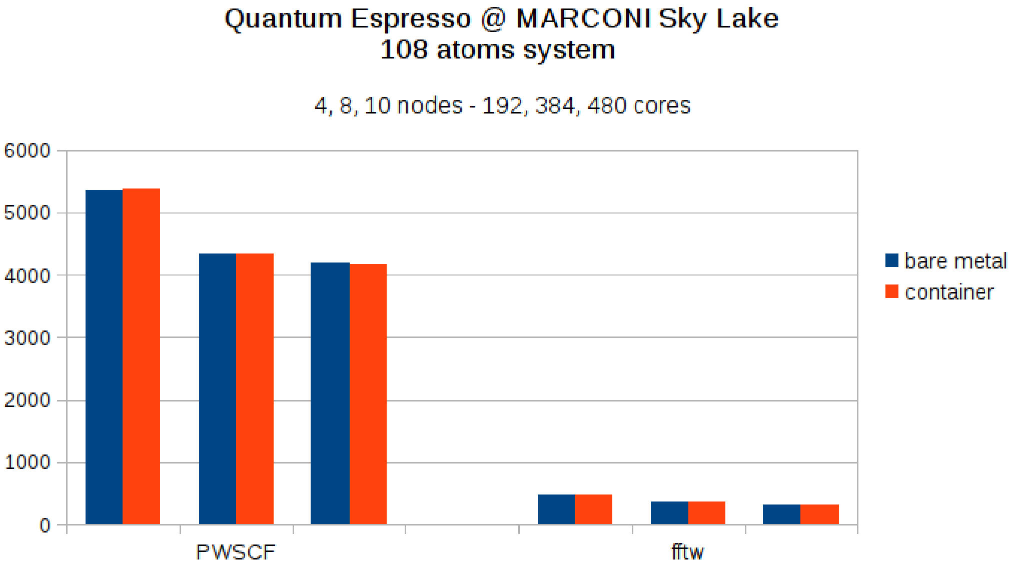


Figure 7.4: Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (fftw) are graphed for a system of 108 zirconium and silicon atoms run up to 10 MARCONI Sky Lake nodes. Both in the container and bare-metal execution, QE code is compiled using Intel MPI compiler.

Test case 2

In this second test, we analyse the performance of Quantum Espresso compiled with Open MPI.

Both in bare metal and in container, any hardware specific flag has been used at code compilation time, in order to improve portability of the container.

The input chosen was a 24 Zirconium and Silicon atoms, with a K-point mesh of 6 x 19 x 13, pure MPI. In the singularity container built, the OFED driver for the Infiniband Network has been installed, together with the application.

The performance comparison is shown in the graph below Fig. 7.5. Once again the total execution time is reported for the entire code (PWSCF) and for the Fastest Fourier Transform in the West (fftw), any calculation was averaged over 5 trials, run on 1, 2, 3 and 4 MARCONI Sky Lake nodes.

The overhead introduced using containerized Open MPI libraries is negligible, since the total execution time of bare metal and container runs are comparable.

Comparing such results with those in Figure 7.3 (same QE input) is clear that QE compiled with Intel MPI are about 5 times faster than compiled with Open MPI. This is widely known from literature and is not affected by the containerization procedure.

The container used for this test is portable. There are not used hardware depend flags, libraries or licenced software.

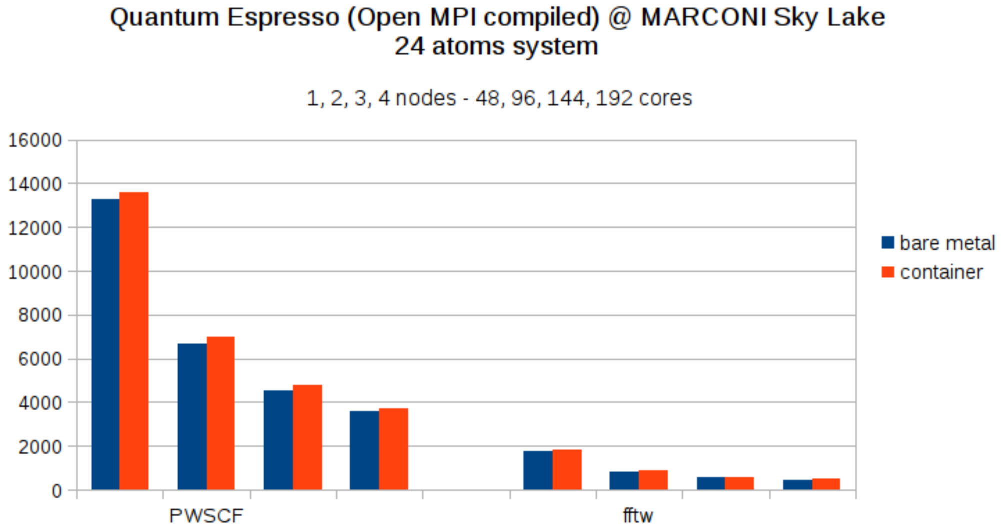


Figure 7.5: Total execution time of the simulation (PWSCF) and Fastest Fourier Transform in the West (fftw) are graphed for a system of 24 zirconium and silicon atoms run up to 4 MARCONI Sky Lake nodes. Both in the container and bare-metal execution, QE code is compiled using Open MPI compiler.

8. HLRS Benchmarks

HLRS operates a supercomputing system Hazel Hen (Cray XC-40) [40], which is used by its academic (German and EU) and industry customers (e.g. Porsche, Daimler, etc.) The system is used for production runs and is therefore quite restricted in terms of access and usage in order to guarantee high security and other policies. Therefore, the HPC-Europa3 virtualization benchmarks were performed on a separate test system, called EXCESS (installed in the frame of a homonymic project). The system consists of 6 nodes (front- and back-ends as well as 4 worker nodes) with a similar architecture as Hazel.

The nodes hardware has the following characteristics:

- node01: 2 * Intel Xeon 5-2693-v2 (Ivy-Bridge) CPU, with 10 cores each, 32 GB DDR3, IB-FDR,
- node02: same as node01 + NVIVDIA-K40 GPU
- node03: 2 * Intel Xeon 5-2680-v3 (Haswell), with 12 cores each, 128 GB SDRAM-DDR4
- node04: Intel KNL Xeon Phi 7210, 64 cores, 94GB DDR4, NVML Intel PCI SSD

The nodes are interconnected with Infiniband FDR 56Gb/s network cards with the following transport mechanisms supported:

- Open UCX
- OpenFabrics Verbs
- Shared memory/copy in+copy out
- Shared memory/Linux CMA
- TCP over IB

The system software of the nodes is served by:

- Nodes 01,04: Scientific Linux 6
- Nodes 02,03: CentOS v. 7.6.1810

For the evaluation, we used nodes 01 and 02 to cover a possibly broad CPU configuration.

8.1. Applications and benchmark scenarios

HLRS is mainly specialized on engineering applications. Approximately a half of all applications running on HLRS resources are served by the CFD (Computational Fluid Dynamics) simulations. Physics applications is the second largest application domain of HLRS (roughly 1/3 of all applications). Due to numerical schema relying on fine-grain time discretization, the major requirement on the infrastructure, as imposed by the majority of the applications, is the high-speed network interconnect. The other requirements include availability of vectorization (e.g. AVX2), low I/O latencies, fast memory bandwidth etc.

For the WP12 benchmarking, a set of the HLRS applications and benchmarks was selected, which is representative for the major workloads running on the HLRS systems. By the way, a similar applications set is used for the procurement activities, whenever a new HPC system shell come in place. The applications set includes:

- Simple communication benchmark (a Ping-Pong test for network latency and bandwidth)
- Universal CFD toolset (OpenFOAM)
- Optimized CFD toolset (Palabos)

- Highly-optimized Multi-Physics application (PACE-3D)
- Data Analytics applications (Word Count and Semantic Spaces)

Below we give a brief description of the applications and benchmarked cases.

8.1.1. Communication benchmark

This is a simple test that aims to measure the time needed to transfer data between the distributed compute nodes via their network interconnect. The test allows to evaluate the network s bandwidth and latencies. Despite being very simple in terms of functionality, the test gives important insights into the functionality of the nodes interconnect. In this test, a message of the fixed or also varying size is passed between each pair of CPU cores of the interconnected nodes. The test allows to check not only the network characteristics, but also evaluate the ability of the communication layer (e.g. of the MPI library implementation) to use all special hardware features to maximize data transfer rates. The HLRS version of the Ping-Pong test is developed in C language.

8.1.2. OPENFOAM

OpenFOAM [41] is probably the most wide-spread software package implementing CFD (but also chemical, solid mechanics, electromagnetic) simulations, which follows the classical CFD approach (Navier-Stokes method). OpenFOAM is a big library of open-source numerical solvers for CFD case studies. A number of useful utilities, e.g. for mesh creation, post-processing and others are provided as a part of the OpenFOAM package as well. For our evaluation, an application investigating aerodynamic properties of the airflow through the object of a complex geometry (motorbike) was selected (8.1). This sort of simulation is a typical CFD analysis task and also used during the training courses offered by HLRS and PRACE. The application included steps of mesh generation for the obstacle geometry, domain decomposition, and numerical solution (limited to initial 100 steps).

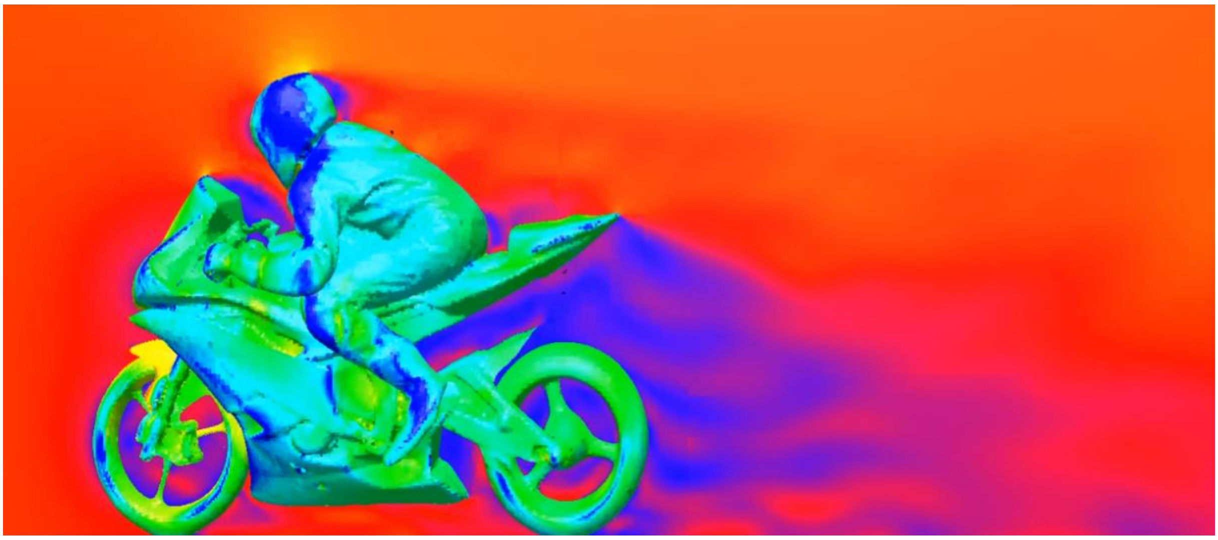


Figure 8.1: OpenFOAM evaluation case: CFD simulation of airflow through a complex geometry OpenFOAM is implemented in C++, uses MPI parallelisation and exposes a network-bound performance pattern.

8.1.3. PALABOS

Palabos [42] is another CFD simulation package, developed by University of Geneva. It specifically targets the lattice Boltzmann method (LBM) for analysis of incompressible fluid problems – a wide-spread modern numerical method that simulates the flow on a lattice with streaming and collision (relaxation) processes, instead of directly solving the Navier-Stokes equations. LBM has a limitation of having application in the low-Mach number flow regime.

Palabos offers a rich-functional framework for development of CFD simulation applications with LBM. Unlike the previously presented OpenFOAM, which focuses more on the universality and neglects some aspects of the performance tailoring, Palabos offers much more optimization options, which however requires more coding efforts from the programmer. Palabos is used by many HLRS customers, including the industrial ones (like Porsche).

The containers evaluation was performed in a similar fashion as for the above-described OpenFOAM, but for a more complex geometry (a car profile, Figure 8.2), in order to evaluate the offered optimization features of the HPC hardware.

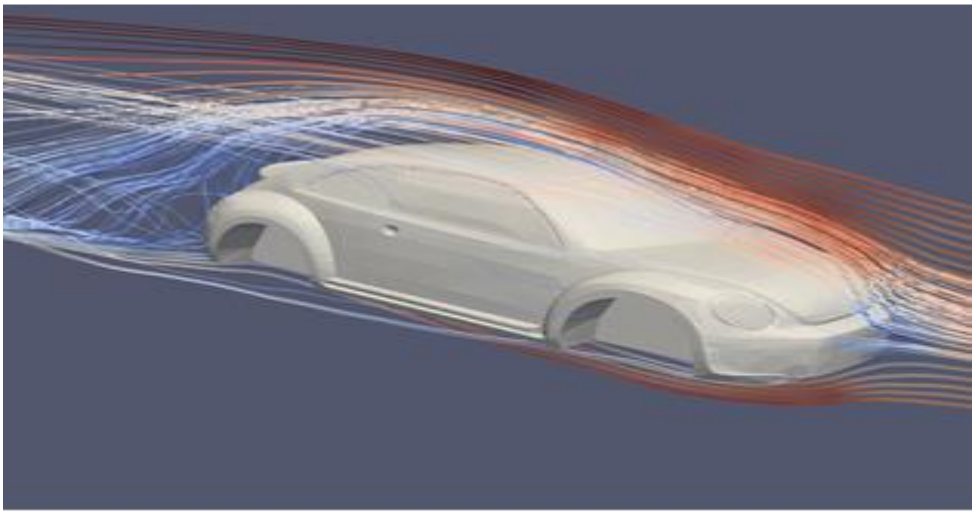


Figure 8.2: Palabos evaluation case: CFD simulation of airflow through a vehicle profile

Palabos is implemented in C++, uses MPI parallelisation and exposes a network- and IO-bound performance pattern.

8.1.4. PACE-3D

PACE-3D is a multi-physics phase-field application developed by Karlsruhe Institute of Technology (KIT). It is MPI-parallel and reaches performance in PFLOPs range. In order to ensure high resource utilization, the application has been optimized to use the majority of features of the modern CPUs (like SIMD-vectorization, e.g. with AVX2) and networks (like RDMA).

The PACE-3D application, which was selected for our evaluation, performs analysis of microstructural material properties during a sintering process – turning loose powders into dense materials, such as can be found in the natural formation of glaciers.

PACE-3D is implemented in C, uses MPI parallelisation and is strongly memory-bound.

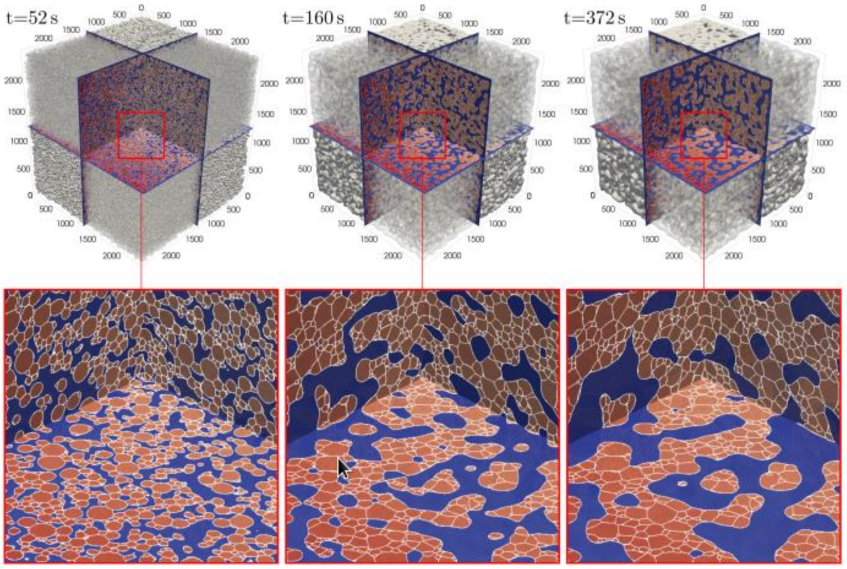


Figure 8.3: PACE-3D evaluation case: Simulation of microstructural material properties during a sintering process

8.1.5. Data analytics workloads

Big Data and Data Analytics have become new HPC application domains since the last 3-4 years. Typical Big Data workloads perform analytics operations that are less computation-intensive than the typical HPC workloads but still require large-scale executions.

For the evaluation, we selected 2 challenging benchmarks – Word Count and Semantic Spaces. Word Count is a typical benchmark that is supplied with Hadoop [43] – the most famous implementation of the MapReduce programming model. Word Count takes as input a large text file and calculates the occurrence number of every word that appears in this text, e.g. for the MPI-3.0 standard specification:

“communication”: 1224

“collective”: 308

For this, a Map is created on every process (running on a parallel system) that includes a word and a number of its occurrences in the text chunk, assigned to this process. In the Reduce phase, the maps of all processes are shuffled and the resulting “global” map is produced.

The 2nd benchmark scenario – Semantic Spaces – comes from the Semantic Web application domain, which is known by its challenges in terms of Big Data processing (large size of data, high dynamics of data and thus many computations needed). Semantic Space technology was selected as one of the most computation-intensive application areas of Semantic Web and also because of the availability of MPI-parallel code (Airhead, see below). The code has won a parallelisation challenge prise of Krakow University of Technology in 2011.

Semantic Spaces aims to statistically derive the context of the textual information contained in the large amount of textual data. Similarly to Word Count, Semantic Spaces analyses the content of large text files, but with a much more challenging goal – to identify the major concepts (i.e. semantic spaces) and analyse (i.e. index) relations between them. In our evaluation, the indexing was performed with the Airhead-SS [44] package, which implements an optimized indexing technique called Random Indexing. With random indexing, it is possible to identify that, for example, the phrase "collective communication" means pretty much the same as "collective information exchange" but very much different from "collective mind", despite of having the word "collective" in both phrases.

Both Word Count and Airhead-SS were implemented in Java and parallelized with MPI (OpenMPI Java bindings were used). Word Count is typically I/O bound, and Airhead-SS is memory-bound.

8.1.6. Container environment setup

For the evaluation we were considering 3 major containerization technologies: Docker, Shifter, and Singularity. Due to the HLRS administrative restrictions, Docker installation was inappropriate due to the required full, unrestricted root access on production systems – this is both to prevent users from accidentally performing actions that may cause issues in production, and is also likely necessarily to satisfy audit requirements where user actions must be audited and access rights must be enforced.

Shifter and Singularity provide same functionalities as Docker, without any of the drawbacks listed above. Using a completely different implementation, they don't require any privilege to run containers, and allow direct interaction with existing Docker containers. When choosing between Shifter and Singularity, we relied on the analysis performed by the deliverable D12.3 [25]. According to it, Singularity offers a similar functionality as Shifter but offers more advantages in terms of support and extendibility on the production system. Therefore, Singularity was finally selected as the basis containerization technology for the HLRS evaluations.

The application container was set up on the basis of a Docker container that is available for OpenFOAM [45] with the characteristics listed in Table 8.1. The initial container distribution already included the most common tools for setting up and running parallel applications. However, some tailoring was necessary to be undertaken before taking the container into production (also listed in Table 8.1).

Table 8.1: Properties of basis and final container

Characteristic	Basis Container	Production Container
OS	CentOS 7.3	
Compiler	GNU	GNU 9.2.0
MPI		OpenMPI 4.0.1
Transport layer	TCP	TCP, Infiniband

Most notably, the support for a high-throughput interconnect (in our case – Infiniband) had to be provided, as the network latency and bandwidth are the major limitation factors for performance and scalability of most of the HLRS applications, including the ones, evaluated in this deliverable. The standard Infiniband drivers had to be installed into the container, such as MXM, HCALL and FCA.

The basic MPI library had also been updated in order to enable Infiniband communication and also to support Java. A support for Singularity was added as well.

The evaluated application benchmarks have been installed into the container in order to enable the execution without binding any external directory (however, the evaluated Singularity allowed this with very low configuration effort).

8.1.7. Benchmarking procedure and results

The goal of benchmarking was to find out how the functional and non-functional properties of the “bare-metal” applications change after porting them into the “container environment”.

The benchmarking methodology that we were following for this was quite straightforward. Firstly, for each of the benchmarked applications, a functional validation was done. The goal of the functional tests was to ensure that the containerized applications can be successfully started and completed and that the produced results were valid. In order to ensure the equality of the comparison, the bare-metal tests were performed in the similar system software environment (compiler, MPI library, etc.) as installed in the container with some minimal deviations due to different OS and low-level system software in the container and on the host.

In the second phase, the non-functional execution properties (most notably – the total execution time) were evaluated. For this, all benchmarks were executed on the Infiniband-interconnected nodes 1 and 2 that were available in the privileged mode (no other jobs that might have disturbed the execution were running at the moment of benchmarks execution). The execution time was measured with the help of “time” Linux command (the “real time” was measured).

All tests were executed 20 and the average values of the execution time were taken. The use of Infiniband was enforced by corresponding directives of the MPI runtime environment. The typical execution command (for n=4 MPI processes on m=2 nodes, as an example) for the bare-metal tests was as follows:

```
mpirun --prefix /opt/centos7/mpi/openmpi/4.0.1-gnu-9.2.0-ib \
-np 4 --host node01,node02       \
--mca btl_openib_allow_ib true   \
/home/mybinary
```

and for the container:

```
mpirun --prefix /opt/centos7/mpi/openmpi/4.0.1-gnu-9.2.0-ib\
     -np 4 --host node01,node02 \
     /opt/singularity/3.1.1/bin/singularity exec \
     /nas_home/hpcochep/Singularity/Containers/openfoam_v19.06_writ
     able /home/mybinary
```

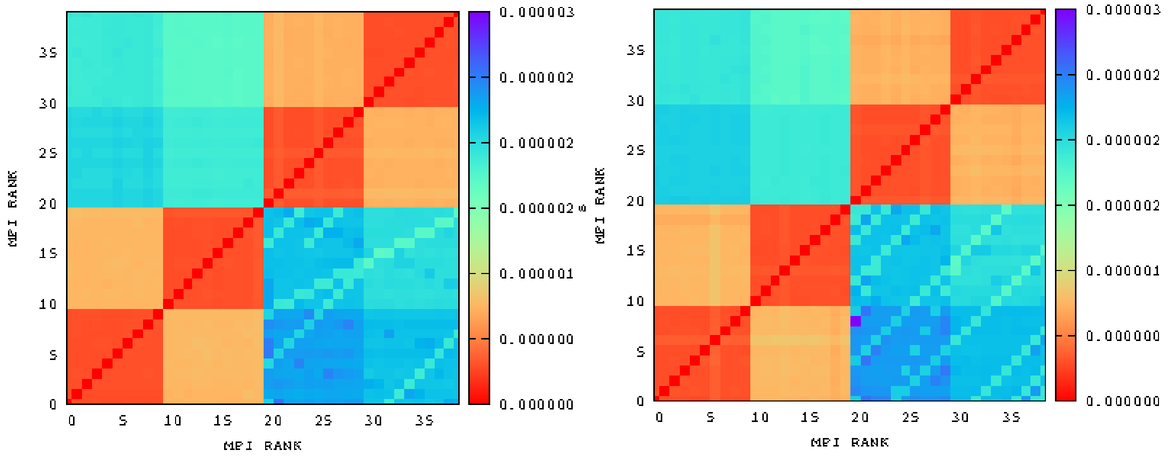
Communication benchmark

In the test, 1 MPI process was launched per core, meaning that for both used nodes, a total of 40 MPI processes were running. Ranks 0-9 (node01) and 20-29 (node02) were running on the 1st CPU socket, and ranks 10-19 (node01) and 30-39 (node02) on the 2nd CPU socket.

The functional results of the tests were the values of i) latencies (Figure 8/4) and ii) bandwidth for the inter-node communication (Figure 8.5). The communication was performed between every core of both CPUs (20 cores per CPU were available, so that each core had to perform 39 communication operations). All tests have passed successfully.
From the analysis of the communication time properties, it can be seen that the containerized application shows a somewhat larger dispersion for shared-memory communication (i.e. communication between the cores of the same CPU socket), which might be caused by the way of

thread handling by the container OS. But the internode communication (through Infiniband) seems to be not impacted by any changes.

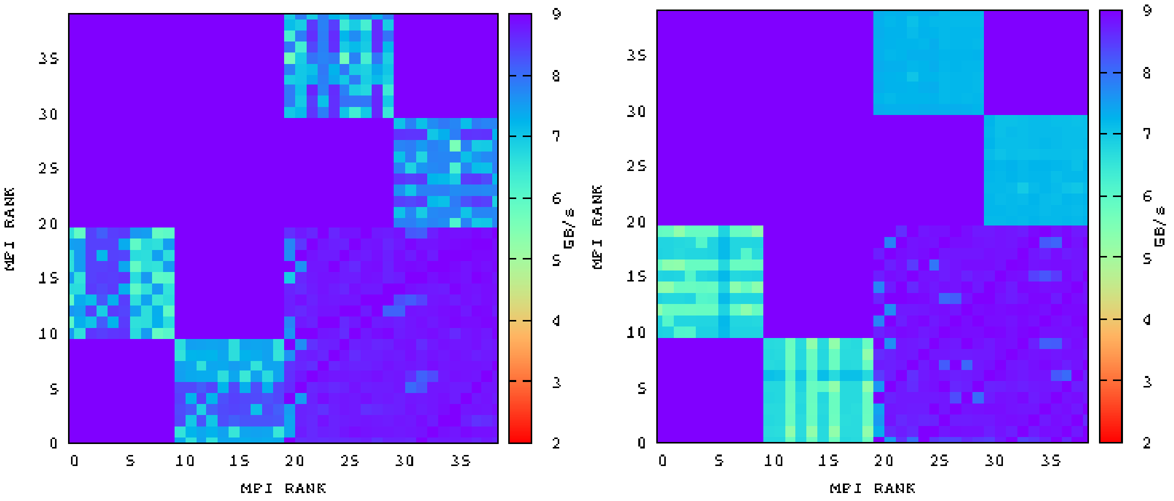
As a consequence, the applications that rely heavily on OpenMP might experience some performance gains or penalties, depending on their communication pattern.



a)

b)

Figure 8.4: Latencies for 10000 exchanges of a 4-byte message between all cores in a) bare-metal, b) container configurations



a)

b)

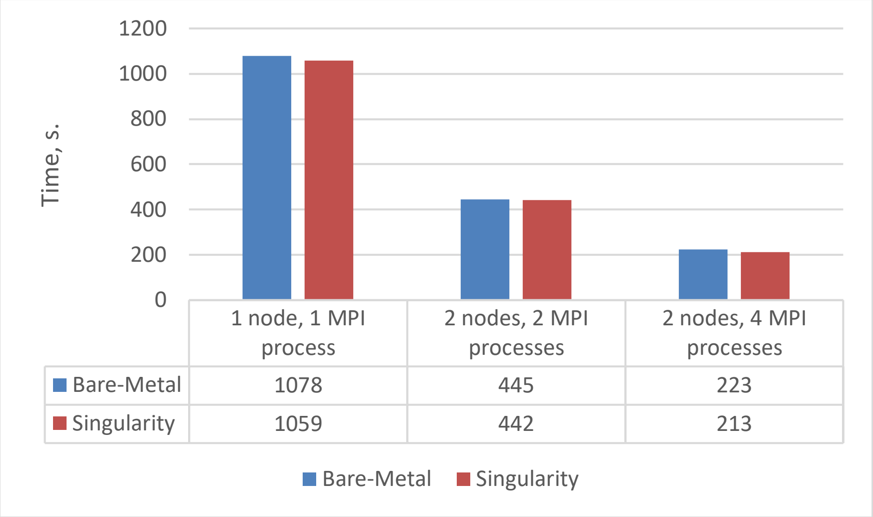
Figure 8.5: Bandwidth for 50 exchanges of a 1-Megabyte message between all cores in a) bare-metal, b) container configurations

OpenFOAM

In order to evaluate the impact of the above-described differences in shared memory communication between the bare-metal and container configuration, OpenFOAM was compiled and run with the instructions to rely as much as possible on the OpenMP communication (inside a node). The configuration with 1, 2, and 4 MPI processes (total number on all 2 used nodes) were evaluated with the maximum possible amount of OpenMP threads per node.

In such a configuration, the effect of running the application in a container environment was positive: **OpenFOAM was able to gain up to 4% of performance improvement when running in a**

container environment with the presumable "better fitting" OS and system software. Figure 8.6 gives more details on the OpenFOAM performance.

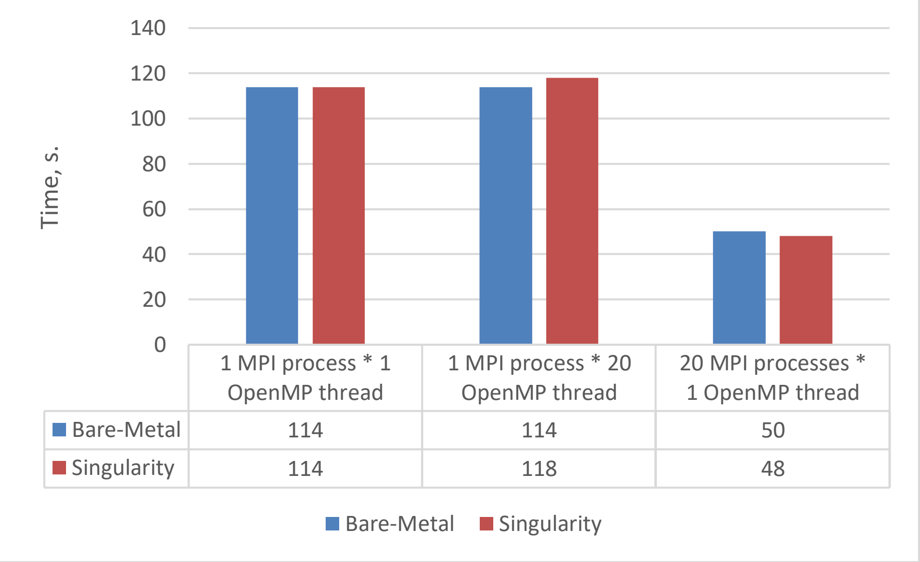


	1 node, 1 MPI process	2 nodes, 2 MPI processes	2 nodes, 4 MPI processes
🟦 Bare-Metal	1078	445	223
🟥 Singularity	1059	442	213

Figure 8.6: OpenFOAM performance in a) bare-metal, b) container configurations

Palabos

Palabos exploits a more computationally-heavy performance pattern and therefore benefits less from the intra-node parallelisation than the above-discussed OpenFOAM. However, even for Palabos, the OpenMP realization in the container offered slightly better performance for the 10 second (physical time) simulated use case (Figure 8.7) for the case of the MPI processes. For the affinity-agnostic OpenMP threads with possible context changes between the threads (which was presumably the case for the OpenMP-based parallelization), the performance might slightly degrade, which was also illustrated by the previous communication benchmark.



	1 MPI process * 1 OpenMP thread	1 MPI process * 20 OpenMP thread	20 MPI processes * 1 OpenMP thread
🟦 Bare-Metal	114	114	50
🟥 Singularity	114	118	48

Figure 8.7: Palabos single-node performance in a) bare-metal, b) container configurations

For the parts which rely on the node-interconnection, the containerized version has shown a similar performance with a slight trend to a slow-down when performing the communication over Infiniband

(Figure 8.8), with an expectation of getting neglectable for the larger parallel configurations (with more nodes and MPI processes involved).

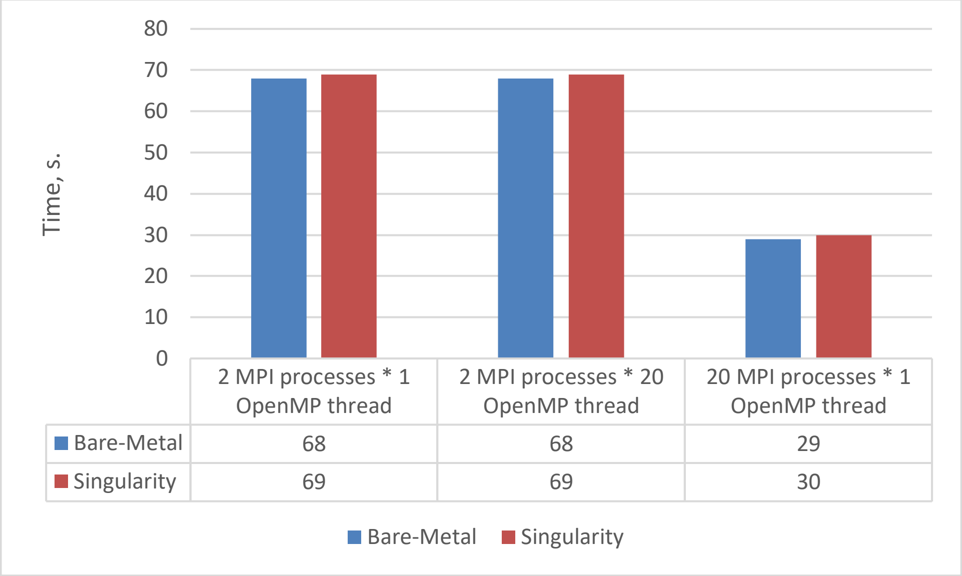


Figure 8.8: Palabos 2-nodes performance in a) bare-metal, b) container configurations

The trend is also clearly visible from the speed-up chart of the MPI-parallel implementation (where the base case is served by a single MPI process running on 1 core, Figure 8.9), as soon as the communication shifts from the shared-memory parallelization to the inter-node one (over Infiniband).

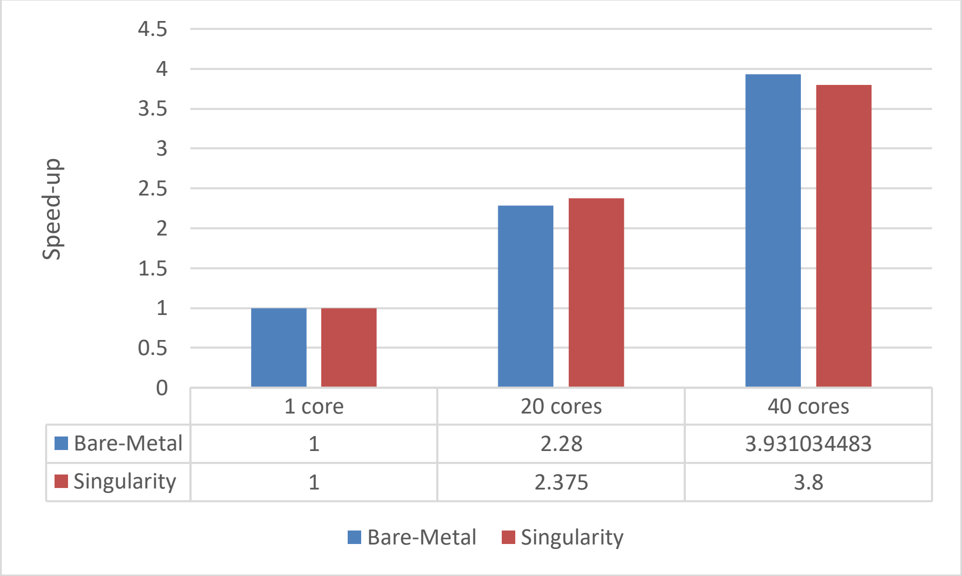
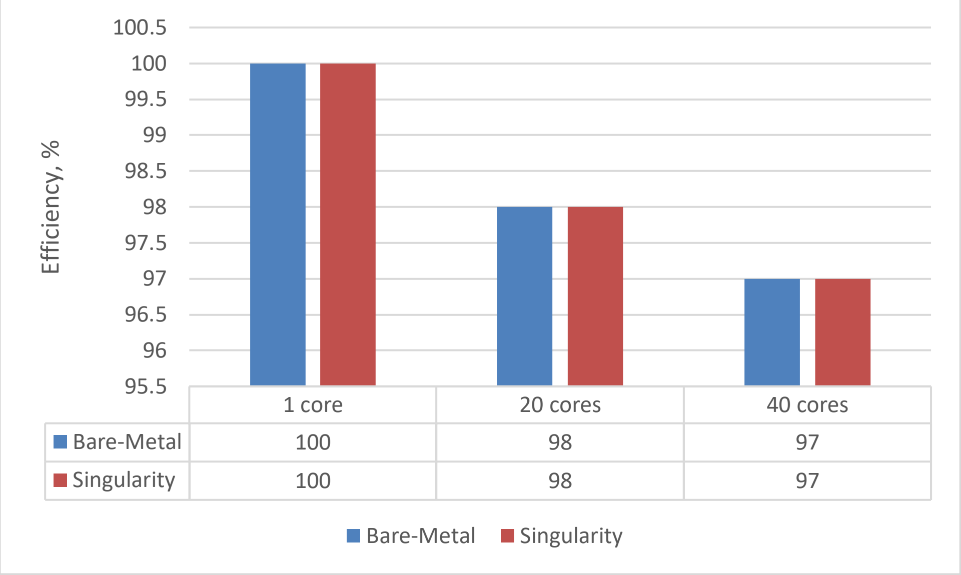


Figure 8.9: Speed-up of MPI-parallel Palabos implementation in a) bare-metal, b) container configurations

PACE-3D

PACE-3D is a highly optimized solver for the sintering process to efficiently solve large domain sizes with long integration times. The high efficiency is achieved by exploiting parallelization at both levels: intra-node (with vectorization intrinsic) and inter-node (with the efficient MPI communication pattern). Only-MPI deployment configurations were tested (there is no OpenMP implementation available).

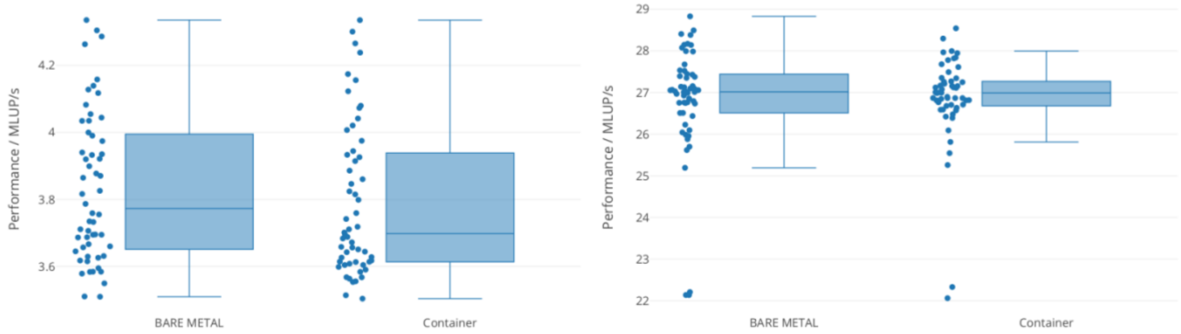
The application was tested on 2 domain sizes: 640×480×640 and 1920×6720×1280 voxel cells, representing a real physical problem of one of the KIT (application developer) customers. We had to stick to these two pre-defined geometries as their tailoring was not easily possible. For this reason, only strong scaling tests were performed (weak scaling had to be omitted due to the case restrictions) and the efficiency (speed-up divided by the number of all used cores) was measured with the same result for both domain sizes – **no performance impact for highly-optimized codes have been observed**.



	1 core	20 cores	40 cores
🟦 Bare-Metal	100	98	97
🟥 Singularity	100	98	97

Figure 8.10: PACE-3D strong scaling efficiency in a) bare-metal, b) container configurations for smaller and larger test domains

In order to evaluate the impact of performance optimization on the containerized application execution, we turned off all optimization features (vectorization, process and frequency pinning, etc.) and run 60 tests with the goal to measure the dispersion of the average execution time (on 2 nodes) between the containerized and the native application. The performance was measured in LUPS (Lattice Updates Per Second) units. The results that we obtained were quite prominent – the dispersion for both small and big domains has slightly decreased (Figure 8.11) and was inappreciably higher for the bare metal version. The average performance value for the big domain didn't change after moving to the container-based execution.



a) b)

Figure 8.11: PACE-3D performance dispersion for a) small and b) big domains

Data Analytics

Unlike the previously presented PACE-3D application, both benchmarked data analytics applications use the compute infrastructure less efficiently. This is partly due to the specific of their workloads (which mainly perform “simple” operations like sorting or filtering) but also partly caused by their implementation (need of Java VM, “fat” object model, etc.).

The Word Count benchmark was performed on the entire MPI-3.0 standard, converted into the text. The total execution time was measured for single- and 2-node configurations without any essential difference in the final performance between the containerized and the native realizations (Figure 8.13).

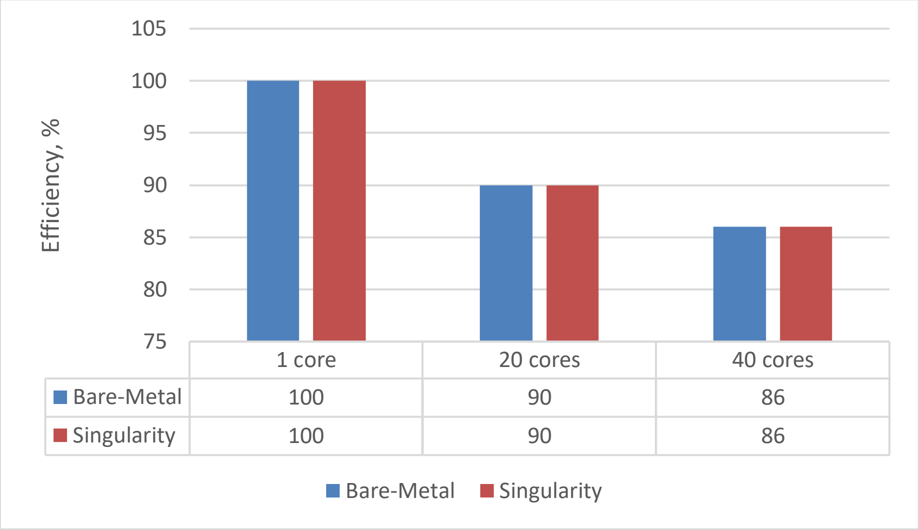


Figure 8.12: Word Count strong scaling efficiency in a) bare-metal, b) container configurations

Similar results were obtained for the Airhead Semantic Spaces applications, which showed a bit better efficiency than Word Count but still no significant difference between the native and containerized versions (Figure 8.13).

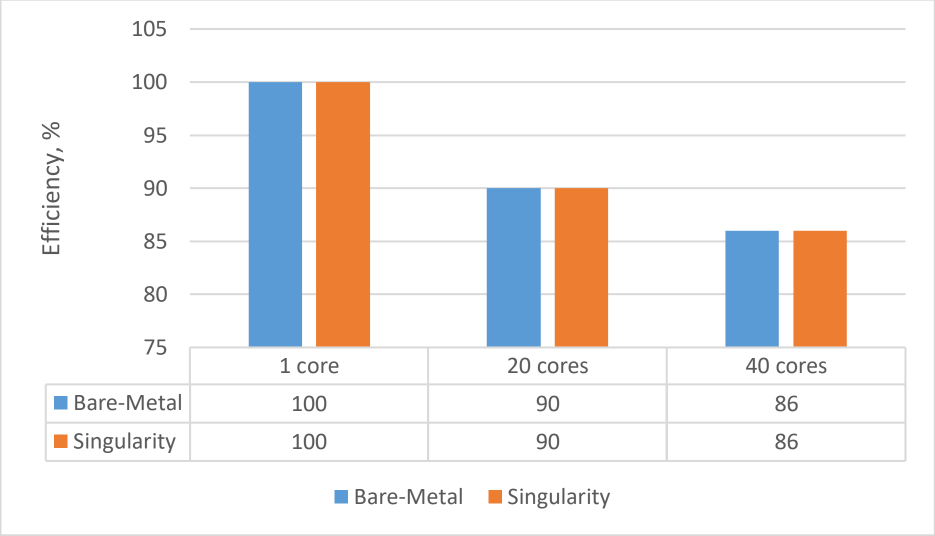


Figure 8.13: Airhead-SS strong scaling efficiency in a) bare-metal, b) container configurations

8.2. Summary

Containers prove being a good option for releasing complete application solutions for engineering domain. Singularity is efficient but at the same time easily configurable tool for providing containers to the end-users.

Engineering applications seem to be quite sensitive against their execution environment, so that the performance might slightly differ even for different versions of the same OS, unless the application is extremely well optimized, which is the case for PACE-3D benchmark, or neglects the performance factor to attain a higher user-friendliness and ease the adoption by the programmers, as it is the case for the benchmarked Java-based Data Analytics applications. For the "mainstream" applications, containerization might offer advantages in terms of performance and efficiency.

A somewhat negative aspect of the container technology is the size of the container. Even if the Linux core is excluded from them, the rest of the systems software and other middleware is relatively "heavy". For example, even for the basic OpenFOAM container, 1.65 GB disk space is needed. If the "write" functionality is enabled, the container needs to be replicated by the users, which can lead to the exceeding disk space very quickly. The standard module environment is many orders of magnitude more efficient in terms of the required disk storage.

At any case, container technologies have proved their general usefulness to the core HLRS expertise area – engineering applications. The offered advantages in terms of the application execution environment customization were highly supported. HLRS interviewed five of its major customers and all indicated their interest in using Singularity in the future. The current HLRS's flagship system Hazel Han is in the process of migration to the newer AMD technology (new supercomputer Hawk). Based on the positive evaluation results of Singularity, this middleware was enrolled for the installation on the new system (to come to production in approximately April-May 2020) and will be provided for all users of HLRS.

9. CNRS Benchmarks

9.1. DIRAC and RAMP

We will use two benchmarks which will allow us to check the performance improvement or degradation regarding the usage of the CPU, GPU, memory and I/O, when running the benchmark code within Singularity container (compared to running it in the same bare machine). In this study, we will only focus on Singularity as it is the most used container solution for scientific workload. It is also the one we provide to our users, especially for running on the GPU farm.

The first benchmark, so called the Dirac benchmark [46], is a simple and fast benchmarking tool which aims to give a good estimation of the CPU resources available on a node. It allows to benchmark from a single job slot to the whole node. It is used in WLCG [47] by the LHCb [48] experiment to quickly benchmark opportunistic resources. That tool returns a number which gives an idea of the CPU power of the machine (the greater is the number, the more powerful is the CPU of the machine).

The second benchmark [49] we will use is a home-made tool based on a toy deep learning challenge organised by French colleagues using the RAMP platform [50] for a school of computing in April 2018. The model implemented is based on a convolutional network called UNet [51]. The whole benchmark is written using the Keras library with a Tensorflow backend. The dataset (2 Gb) is divided into 12 000 images for the training, 4 000 for the validation (during training) and 4 000 images for the final evaluation of the model. This benchmark is intended to be run on GPU, but could also be run on CPU (if there is no GPU on the node). The main metric is the execution time, and the application is memory-, IO- and GPU-bound.

For our benchmarks, we will compare the execution time, but also the usage of the memory and the I/O as we will be able to get these numbers from the report from our batch system, which is Univa Grid Engine. It is worth noticing that whatever the batch system used, it should not impact at all the benchmark results.

9.1.1. Benchmarking environment

We have run the two benchmarks on fully dedicated worker nodes to avoid any interference with others jobs. We had three dedicated worker nodes: a "common" one with a CPU only (32 cores, 100 GB), one with a NVidia K80 GPU (16 cores, 130 GB), and the last one with a NVidia V100 GPU (20 cores, 190 GB).

For both benchmarks, we prepared simple scripts to launch the benchmark code on the dedicated worker nodes through the batch system. We also used the exact same Singularity image for DIRAC and RAMP, which was the one prepared for the RAMP (GPU benchmark). The idea was to have the same environment as much as possible, so we had mostly only the name of the worker node and the benchmark code to update before each new test.

We have also done test with four Singularity versions for the Singularity runtime: 2.6.1, 3.0.3, 3.1.1 and 3.3.0. We chose those versions for the following reasons:

- 2.6: last supported version of Singularity 2.X; not compatible with Singularity 3.X but still widely used.

- 3.0.3 and 3.1.1: versions of Singularity 3.X that were in production at the CC IN2P3 at the time when we did that study.
- 3.3.0: last stable version of Singularity at the time when we did that study, so we wanted to try it and check if performances were improved.

The results obtained when running in the Singularity container were compared to the reference which is running the benchmark code in the exact same worker node (without using a Singularity container).

Here is the software environment used:

- the worker nodes are running CentOS Linux release 7.6.1810 (Core)
- for the GPU worker nodes, we used the following libraries versions: CUDA 9.2.148 and CUDnn 7.3.1.20

The Singularity image was built from a CentOS 7 docker image (the base Linux system was of the same version as above after a proper update), and we installed the required CUDA and CUDnn libraries inside. We used Singularity version 3.0.3 to build the image. The Singularity image's size is 4GB.

We compiled TensorFlow 1.11 against these libraries and built a Python module (wheel).

9.2. More details about running RAMP (GPU benchmark)

In this case, here is the two ways to run the benchmark: bare metal worker node (which will stand as the reference) and within a container instantiated in the same worker node.

In a bare metal worker node

We setup a conda environment (Python 3.6) in which we installed Keras and the compiled TensorFlow wheel module. This installation is on a shared filesystem and thus visible from all the worker nodes. The CUDA and CUDnn libraries were the ones installed on the worker nodes.

For the reference, we then only have to setup the conda environment and the CUDA libraries path and then run the RAMP benchmark. It returns a time execution, and thanks to our batch system reporting, we could also gather memory and I/O consumption. These will be the three metrics we will check with that benchmark.

In a Singularity container

In this case, we instantiate a Singularity container, in which we bind-mount the shared filesystem. We set up the same conda environment as above but the CUDA libraries installed in the container. Then, in this case, the benchmark runs the TensorFlow code on the GPU by going through the container as it will use the CUDA and CUDnn libraries installed inside. We will then mostly check whether adding the container layer increases the time execution (this is the main purpose of this benchmark) by degrading the performances.

Two different setups have been studied. In the first one, we read the Singularity image from a shared filesystem, and in the second we will read it directly from the worker node memory. In the first case, we will then also take into account possible network latency in the overall overhead.
Moreover, for both cases, the dataset required to run the RAMP / GPU benchmark was stored in the memory of the worker node (i.e. in /dev/shm).

9.2.1. Results of RAMP (GPU benchmark)

As discussed above, in most of the following plots, the reference value for a given metrics is obtained by running the benchmark code in the bare metal, dedicated worker node (to avoid any interference with other jobs, or any tasks running on the machine).

The comparison will then be done against results obtained when running the benchmark code in a Singularity container, considering four possible Singularity versions for the Singularity runtime (see above).

The Singularity image will be the same for all tests and benchmarks and has been built with Singularity 3.0.3.

Wallclock and CPU/GPU time

<u>Environment 1:</u> NVidia V100 GPU worker node with the Singularity image in memory

The wallclock time is the whole time the job has spent in the machine. The CPU/GPU time is the time the CPU/GPU is effectively being used. The latter is usually less than the wallclock time as the wallclock time will also include time for setting up the prerequisites for the jobs and potential access time to data while the job is running.

The CPU time is returned by the benchmark. Hence, we can get the overall overhead with the following formula:

$$overhead\ (t)\ = wallclock\ time\ (t)\ -\ CPU\ time\ (t)$$

Instantiating a Singularity container can require some time (to read the image, then to instantiate the container itself). Following that, we can check below the potential overhead of instantiating a Singularity container:

$$overhead_{singularity}\ (t)\ = overhead\ (t)\ -\ overhead_{baremetal}\ (t)$$

Figure 9.1 shows the resulting overhead (in seconds) for the various Singularity runtime versions used compared to the reference. Instantiating a container adds a slight overhead, 1 to 2 seconds, to the job set up time. In that figure, we ran 130 tests on a NVidia V100 GPU (26 tests for each Singularity version, and the same number without Singularity. The Singularity image and the data required by the benchmark have been copied directly to the memory (RAM) of the worker node to minimize the I/O to access them (they were writing to /dev/shm).

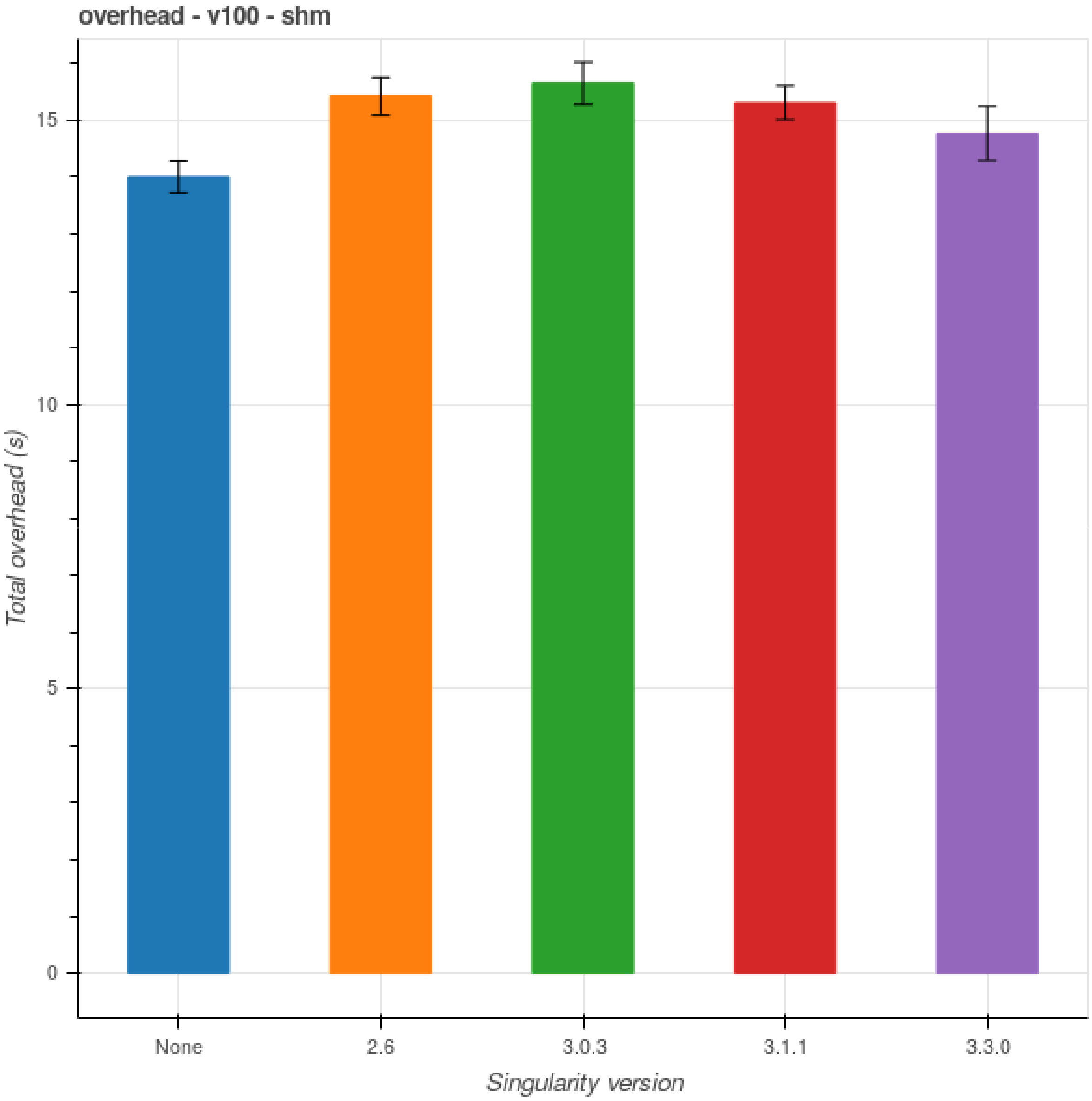


Figure 9.1: Overhead observed with various Singularity runtime versions for the RAMP / GPU benchmark. The data and the Singularity image were stored in the RAM.

The errors reported on Figure 9.1 (and the Figures below when reported) are the statistical errors only, and computed as follows:

$$\varepsilon\ (mean\ error)\ =\ \sigma\ /\ \sqrt{n},$$

where σ is the standard deviation of the distribution of the measurements and n the number of measurements. This can only be computed if the distribution of the measurements follows a Gaussian distribution, which should be the case here. In all tests, n, the number of measurements (tests) is 26.

According to Figure 9.1, there is a small overhead running inside a container, of about 1 to 2 seconds, coming from the required time to instantiate and set up the container. In order to have a better idea of it, one can compute the overhead as an increase, in percent, of the overhead reference value, just as follow:

$$overhead\ (\%)\ =\ (Sing\ -\ None\ )\ /\ None\ ,$$

where Sing represents the overhead value obtained for a given Singularity version and None when running the benchmark code on bare metal machine.

According to Figure 9.2, the overhead for the various Singularity runtime versions is about 10% or so for Singularity 2.6.1, then increasing to about 12% for Singularity 3.0.3, then decreasing to about 5% for version 3.3.0, which shows that the latest version was well improved.

The overhead discussed above mainly comes from the configuration requested for the container. For instance, in our case, in order to see the GPU within the container we have to add the –nv option. More complex the configuration of the container, longer it will take to be instantiated. This is something we will confirm with the use of the Dirac benchmark (for which, even though we are using the same container, the configuration is much more simple).

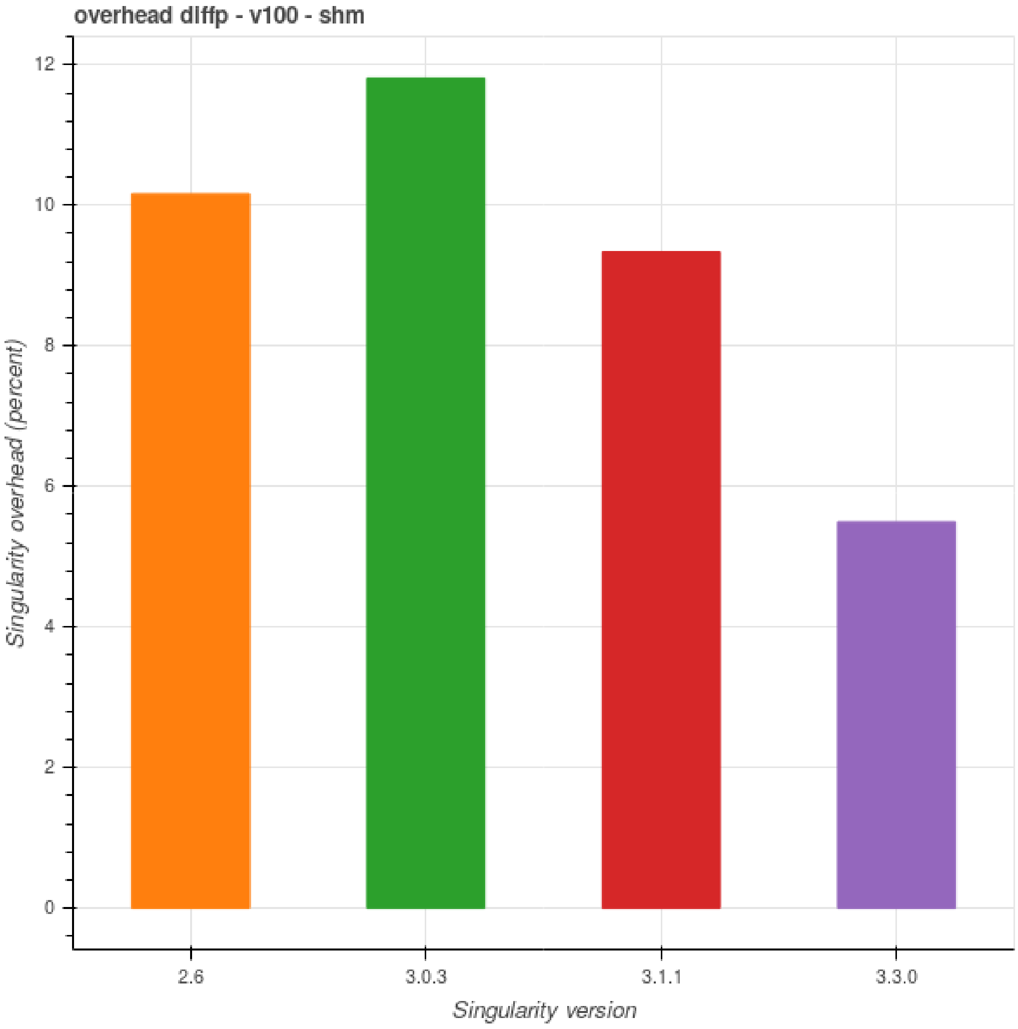


Figure 9.2: Overhead in percent for the various Singularity runtime versions for the RAMP / GPU benchmark. The data and the Singularity image were stored in the RAM.

Environment 2: NVidia V100 GPU worker node with the Singularity image in a shared filesystem

We have redone the same measurements, but with the Singularity image was stored on a shared filesystem. In that new configuration, we can have an idea of the overhead coming from reading the Singularity image in order to instantiate the container, as memory access is much faster than from the shared filesystem.

Figure 9.3 shows the overhead observed, in seconds (left) and in percent (right), just as discussed before in Figure 9.1 and Figure 9.2. Comparing Figure 9.3 to Figure 9.1, one can note a slight 0.5s increase for reading the required data from the image to instantiate the container. This is an interesting result, as it shows that even a shared filesystem (without providing excellent I/O) can be used as a Singularity image repository.

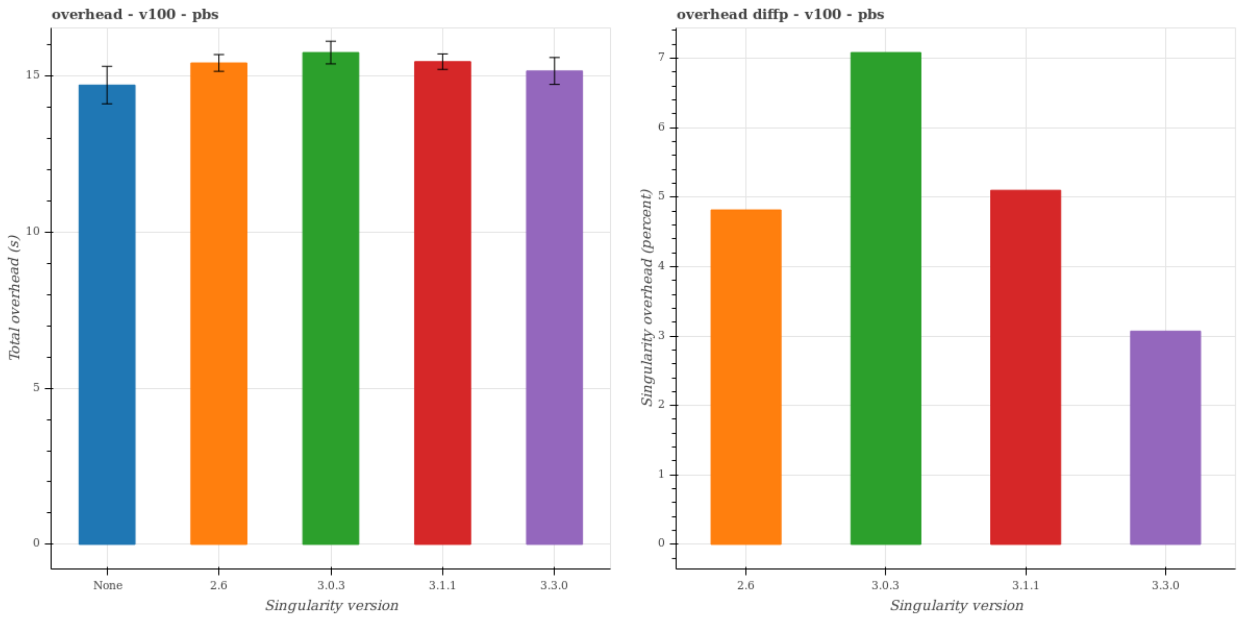


Figure 9.3: Total overhead (on the left) and overhead in percentage (on the right) observed for the various Singularity runtime versions for the RAMP / GPU benchmark. The Singularity image was here stored on a shared filesystem.

Comparing Figure 9.2 and 9.3, one can notice the exact same pattern evolution of the increase, when going from Singularity 2.6.1 to 3.3.0, with the latest version being quite optimized compared to the others. This pattern will also be there when using another kind of worker node (Figure 9.4).

Environment 3: NVidia K80 GPU worker node with the Singularity image in a shared filesystem

We have redone all the previous tests on a NVidia K80 GPU. Figure 9.4 show the results obtained when storing the Singularity image in the shared filesystem. We of course do not expect new results, but they have to be compatible with what have been seen previously, which is the case.

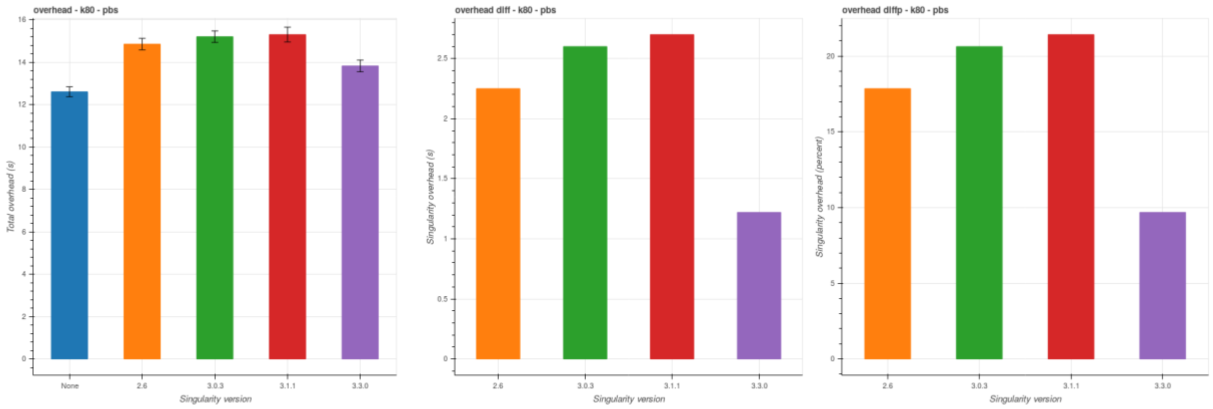


Figure 9.4: Overhead observed for the various Singularity runtime versions for the GPU benchmark. The data and the Singularity image were stored in a shared filesystem. On the left: total overhead, including bare-metal (blue bar) In the middle: singularity overhead in seconds. On the right: singularity overhead in percentage

This overhead difference has to be set in context: it represents a difference of about 15s for the total duration of the job, which is about 570s. This difference is hidden by the error margin of the total duration time average and is actually not noticeable, as seen in the next section.

Time execution

Environment 1: NVidia V100 GPU worker node with the Singularity image in the memory

The next figure show the total duration time average for each version of Singularity. The figure on the right shows the difference (in percent) for each version compared to bare metal. As we can see, the difference is in the error margin and not significant. Hence, we can conclude than even if Singularity add an overhead, the time added is not significative enough to impact the total job duration time.

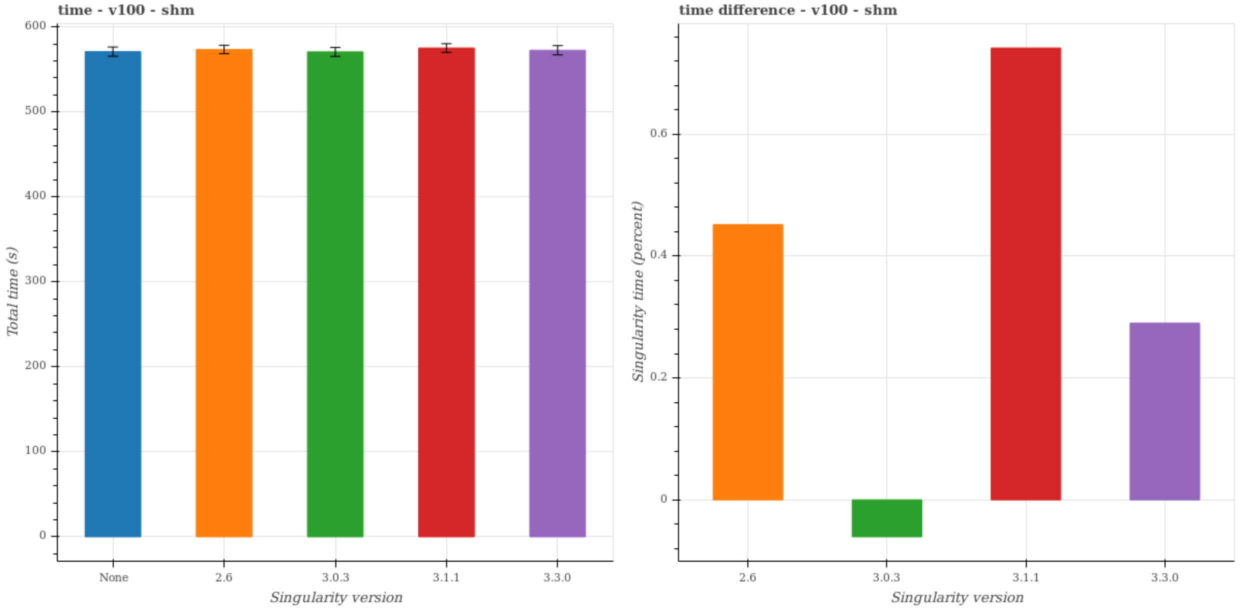


Figure 9.5: Total time of the job (overhead and calculation time) (left) and difference in percent between the time when using different version of Singularity compared to bare metal (right)

Figure 9.5 show the time execution of the benchmark (CPU/GPU time). As one can see, running inside a container does not add any overhead and all the measurement are compatible within the statistical errors. One can then conclude that once instantiated, the container does add any overhead on the performances of the task.

So we have shown that Singularity adds a slight overhead, but this overhead is to small to impact in a significant way the total time duration. The second benchmark will be used to check if the overhead is depedant of the size of the job.

Memory usage

Back to environment 1, we will now check the memory usage. Our batch system allocates memory for each job and then returns how much percent of that memory was actually used by the job. Hence, we can check if using Singularity change the total memory consumption.

Figure 9.6 shows that the average memory utilization is almost always the same. It shows that using Singularity does not impact how the job interact with the worker node to use the RAM.

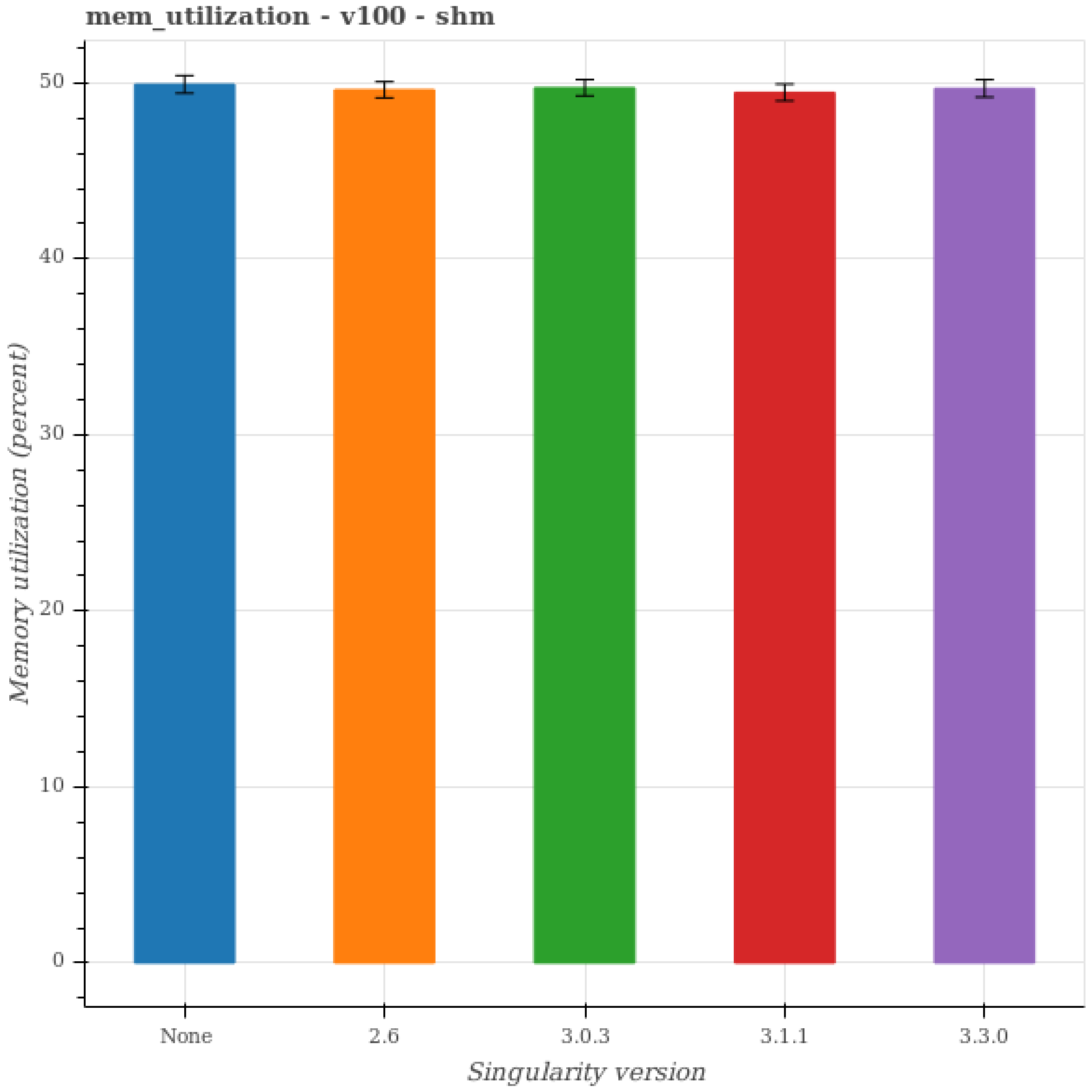


Figure 9.6: Memory usage (percent of the allocated memory for the job) of the benchmark depending on the various Singularity runtime used.

9.3. The Dirac benchmark

The previous benchmark was more dedicated on the following metrics: time execution, memory and I/O usage. We could then check whether using Singularity adds some overhead on these metrics.

The Dirac benchmark, as discussed previously [46], [47], [48] is more focused on checking the CPU power of a machine, and returns a number which represents it: the greater the number is, the more powerful the worker node is. The benchmark gives a fast approximation of the HS06 benchmark, which is the reference benchmark in High Energy Physics [52]. It does not have any unit, but serves as an unit to compare the CPU power of the worker nodes.

Figure 9.7 shows the results of running the Dirac benchmark in various setups: without Singularity, and with 2 versions, 2.6.1 and 3.3.0.

This time only two versions of Singularity were tested: 2.6 (as the last supported 2.X version) and 3.3.0 (as the latest released and the most performant version of Singularity 3.X at the time of the test). Testing more versions would not have given more information than what the RAMP / GPU benchmark returned.

We can see a small degradation of the performances with Singularity 2.6.1, but no significant one with Singularity 3.3.0. This test can be compared to Figure 9.3 which shows the execution time of the previous benchmark (so the CPU/GPU time). On Figure 9.3, all measurements were compatible within the statistical errors. However, the mean value for Singularity 2.6.1 was a bit greater than for 'none' and Singularity 3.3.0, which could indeed indicate that the performances for that version were a bit degraded. More statistics are required on Figure 9.7 to reach a better conclusion, as the statistical errors remains large enough to overlap. However, based on the Dirac benchmark, the difference between 2.6.1 and 3.3.0 remains low, 12.1 compared to 12.3, which in any case seems to indicate a quite small degradation of the performances.

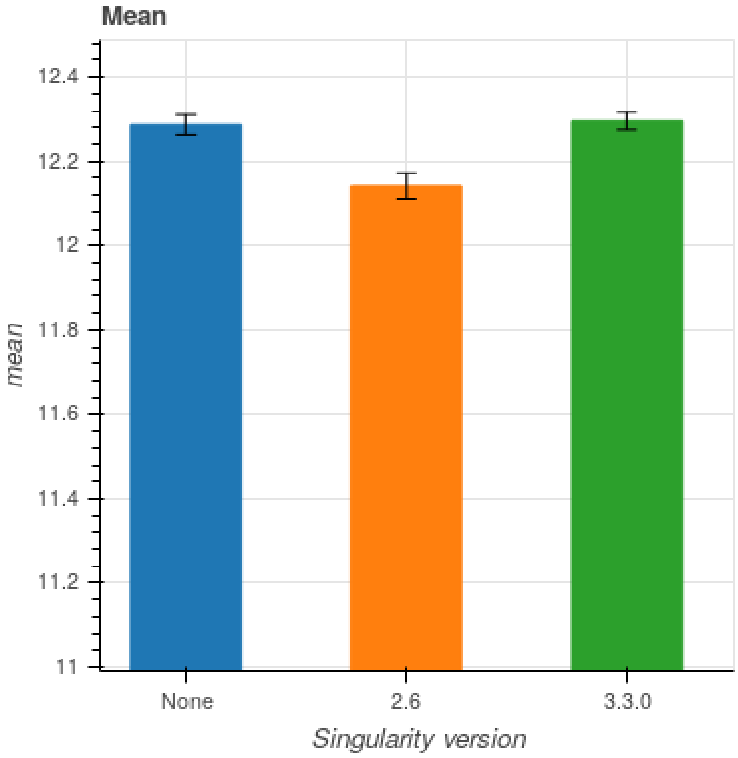


Figure 9.7: Benchmarking of the CPU power of a worker node using the Dirac benchmark, without and with Singularity.

9.4. Summary

We used two benchmarks to measure possible degradation of using a Singularity container towards several metrics: CPU time, memory, I/O. We also compare several Singularity runtime versions to each other, and to the reference, which is running without Singularity.

We observed a small overhead at the start of the job, due to reading the image and setting up the container. In our case the overhead was 1 to 2 seconds, and amounts to about 10% of the time required to set up the benchmark. We moved the Singularity image from the memory of the worker node to a shared filesystem (NAS) and only observe less than 1 s extra overhead, which seems to indicate that reading the image does not account much in the overall overhead of using Singularity.

We compared several Singularity runtime versions, and mostly noticed a quite significant improvement from 2.6.1 to 3.3.0. This overhead was noticeable only for the bigger jobs; for a smaller, simpler one there was no perceptible overhead.

The Dirac benchmark also seems to indicate a slight performance degradation when using Singularity 2.6.1, but it seems to disappear with 3.3.0. Note that the difference between the numbers obtained from the benchmark for these two versions is small, so that the degradation should also be small, and even negligible. This appears to be compatible with the time execution checks done with the RAMP benchmark, which did not exhibit anything significant. Nothing has been observed on the memory usage or on the I/O.

10. Conclusions

In this deliverable, the performance of selected applications was examined by comparing bare metal versus containerized one. The applications' selection was based on popularity in HPC centers and the factors that affect their performance. Docker, Singularity, Shifter containers were also compared. In all cases, Singularity was tested. On some sites, where Docker and/or Shifter is available, they also were tested.

There are two main ways to use of containers. A fully portable container that runs on all types of hardware, at least of the same architecture (x86), or container as a portable development environment that usually needs to be adjusted and recompile the applications inside container taking into account specific systems characteristics.

A fully portable container with precompiled application suffers from performance degradation. It is explored in detail with GROMACS and Tensorflow applications. Compiling the application using high end hardware, minimizes the performance degradation but this makes container not backwards portable i.e. runs on older hardware.

On the other hand, using the container as development environment, compiling the application for specific hardware has in general no effect on the performance. In this case, especially with single node runs –typically using POSIX threads or OpenMP, containerized applications have the same performance than bare metal executions. When more than one node have to be used - this typical implies the use of MPI – performance behaviour is more complicated. Usually, HPC machines have special network hardware of various types, like Infiniband and Omnipath. These special interconnects need the appropriate drivers to be present when MPI is compiled. Except the extra effort to install these drivers inside the container image – assuming that they are publicly available, another complication arises when one moves for example from Infiniband to Omnipath. The procedure of drivers installation and MPI recompilation, should be repeated. In both cases, one has to implement the possible configuration changes for these interconnects inside the container. Although this is possible, for example for administrators who have the knowledge of hardware and possible additional configuration, seems impossible for the application end user. The overhead on the time execution is not significant and does not perceptibly impact the total job duration.

11. References

[1] "http://www.gromacs.org/".

[2] "https://repository.prace-ri.eu/ueabs/GROMACS/1.2/GROMACS_TestCaseA.tar.gz".

[3] J. Dongarra, "Proceedings of the 1st International Conference on Supercomputing," Springer-Verlag, London, 1988.

[4] http://www2.mmm.ucar.edu/wrf/index.php.

[5] https://easybuild.readthedocs.io/.

[6] "'Nanos ++ | Programming Models @ BSC'. [Online]. Available: https://pm.bsc.es/nanox [Accessed: 24-Oct-2019]".

[7] "`Mercurium | Programming Models @ BSC'. [Online]. Available: https://pm.bsc.es/mcxx.".

[8] "M. Garcia, J. Labarta, J. Corbalan, 'Hints to improve automatic load balancing with LeWI for hybrid applications', Elsevier Journal of Parallel and Distributed Computing, vol. 74, no. 9, pp. 2781–2794, Sep. 2014".

[9] "Extrae | BSC-Tools'. Available at: https://tools.bsc.es/extrae".

[10] "Etcd is a distributed key-value store: https://github.com/etcd-io/etcd".

[11] "All the details about Docker's network deployment with Etcd: https://docker-k8s-lab.readthedocs.io/en/latest/docker/docker-etcd.html".

[12] "E. Casoni et al., "Alya: Computational Solid Mechanics for Supercomputers," Archives of Computational Methods in Engineering, vol. 22, no. 4. pp. 557–576, 2015.".

[13] "G. Houzeaux, M. Vázquez, R. Aubry, and J. M. Cela, "A massively parallel fractional step solver for incompressible flows," Journal of Computational Physics, vol. 228, no. 17. pp. 6316–6332, 2009.".

[14] "M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Aris, D. Mira, H. Calmet, F. Cucchietti, H. Owen et al., "Alya: towards exascale for engineering simulation codes," 2014.".

[15] "https://repository.prace-ri.eu/git/UEABS/ueabs".

[16] "Official web page of NPB: https://www.nas.nasa.gov/publications/npb.html".

[17] "[J. H. R. F. vanderWijngaart, "NAS Parallel Benchmarks, Multi-Zone Versions," 2003.".

[18] "H. Jin and R. F. Van der Wijngaart, "Performance characteristics of the multi-zone NAS parallel benchmarks," 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.".

[19] "Oleksandr Rudyy, Marta Garcia-Gasulla, Raül Sirvent, Giuseppa Muscianisi, "HPC-Europa3 - D12.5 - Workload collocation based on container technologies to improve isolation," Jul. 2019.".

[20] ""B2SHARE." [Online]. Available: https://b2share.eudat.eu/records/62310f7e98dd42789ceabc48464251c6. [Accessed: 26-Jul-2019].".

[21] "Compiler directive of the OpenMP programming model to parallelize loops.".

[22] "M. A. Heroux, J. Dongarra, and P. Luszczek, "HPCG Benchmark Technical Specification." 2013.".

[23] "Official web page of HPCG: https://www.hpcg-benchmark.org/index.html".

[24] O. R. e. al., """Containers in HPC: A Scalability and Portability Study in Production Biological Simulations," in IPDPS2019: Proceedings of the Internatiional Parallel and Distrubuted Processing Symposium".

[25] “Giuseppa Muscianisi, Raül Sirvet, Marta Garcia, Oleksandr Rudyy, Niall Wilson, Atte Sillanpää. Ari-Matti Saren, Simone Marocchi, “HPC-Europa3 - D12.3 - Using container technologies to improve portability of applications in HPC,” CINECA, May 2019.”.

[26] “https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide”.

[27] N. W. e. a., “Container as a service analysis report”.

[28] Q. E. https://www.quantum-espresso.org/.

[29] https://www.tensorflow.org/.

[30] “https://hub.docker.com/r/tensorflow/tensorflow/”.

[31] “Krizhevsky, A., Sutskever, I., Hinton, G. E.: Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, 1097–1105 (2012)”.

[32] “Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. Computer Vision and Pattern Recognition (CVPR), 2015.”.

[33] “Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the Incep-tion Architecture for Computer Vision. https://arxiv.org/abs/1512.00567 (2015)”.

[34] “K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. https://arxiv.org/abs/1512.03385 (2015)”.

[35] “Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. https://arxiv.org/abs/1409.1556 (2014)”.

[36] h.-n. ImageNet dataset Homepage.

[37] Q. E. download:http://www.quantum-espresso.org/download.

[38] “https://www.quantum-espresso.org/resources/benchmarks”.

[39] “Giuseppa Muscianisi, Raül Sirvet, Marta Garcia, Oleksandr Rudyy, Niall Wilson, Atte Sillanpää. Ari-Matti Saren, Simone Marocchi,, HPC-Europa3 - D12.3 - Using container technologies to improve portability of applications in HPC,” CINECA, 2019.

[40] https://www.hlrs.de/systems/cray-xc40-hazel-hen/.

[41] “OpenFOAM, https://www.openfoam.com, the ESI distribution of”.

[42] https://palabos.unige.ch/.

[43] https://hadoop.apache.org/.

[44] https://code.google.com/archive/p/airhead-research/wikis/RandomIndexing.wiki.

[45] docker://openfoamplus/of_v1906_centos73.

[46] “https://github.com/DIRACGrid/DB12”.

[47] “http://wlcg.web.cern.ch/”.

[48] “http://lhcb.web.cern.ch/lhcb/”.

[49] “https://gitlab.in2p3.fr/brigaud/prace-ramp-astro-benchmark/tree/master”.

[50] https://www.ramp.studio/.

[51] https://arxiv.org/abs/1505.04597.

[52] “https://w3.hepix.org/benchmarking.html”.

[53] “https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide”.