Annales Mathematicae et Informaticae
58 (2023) pp. 133–146
DOI: 10.33039/ami.2023.08.014
URL: https://ami.uni-eszterhazy.hu

Three level benchmarking of Singularity containers for scientific calculations

Péter Polgár, Tamás Menyhárt, Csanád Bátor Baksay, Gergely Kocsis, Tibor Gábor Tajti, Zoltán Gál[*]

University of Debrecen, Faculty of Informatics,
Debrecen, Hungary
kocsis.gergely@inf.unideb.hu
tajti.tibor@inf.unideb.hu

Abstract. In this work we present our results from benchmarking Singularity containers running scientific calculations intended for HPC at three levels of complexity, comparing them against native software environments. Our investigations were run on up-to-date hardware and the latest available software as of early 2023. To get a more detailed picture, we examined three different system aspects: we ran separate benchmarks on CPU, memory, and I/O intensive operations. These aspects were tested with microbenchmarks, and at a high level – an HPC workload pipeline which intensely loaded each aspect. As a result of our investigations we show that Singularity containerization continues to provide as good, or in some cases even better performance indicators as in a native environment – with the substantial added benefits of application flexibility and portability.

Keywords: Container benchmarking, Singularity, HPC

AMS Subject Classification: Computer Science 68U01, 68U99

1. Introduction

The tools and methods of scientific computing undergo continuous progress, providing opportunities for researchers to achieve more sophisticated and increasingly

[*]This work has been supported by the project TKP2021-NKTA of the University of Debrecen. Project no. TKP2021-NKTA-34 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the TKP2021-NKTA funding scheme.

Submitted: July 28, 2023
Accepted: August 7, 2023
Published online: September 9, 2023

accurate results using better and better facilities. In the meantime, however, history has shown that following new trends in tools and methodologies very often leads to software compatibility issues. It is also a common requirement to be able to run calculations or simulations in differing hardware and software environments. Today's answer to these issues is the use of containerization [3]. By the use of containers, researchers can build uniform software environments, sealed in portable disk images files, and run their simulations or calculations without any regard to the details of the real host's software setup. From this perspective, containers work similar to virtual machines, however, most findings have shown that they operate with much smaller overhead, increasing their efficiency. In the case of scientific calculations, Singularity (nowadays known as Apptainer or SingularityCE – of which we used the latter) is the most common containerization solution used to overcome incompatibility issues between environments. To prove this evident and conventionally accepted statement, we have checked the first 172 HPC providers from the Top500 [15] list to find out what containerization technologies they support. As one can see in Figure 1, 48 out of 172 observed providers published this information on their homepages. Note that while 3/4 of them support only Singularity and/or Apptainer, there are only about 10% which support other containerization technologies, but not Singularity or Apptainer. However, the question continues to arise in an ongoing manner, whether containerization has a performance impact on calculations, and if yes, of what sort?

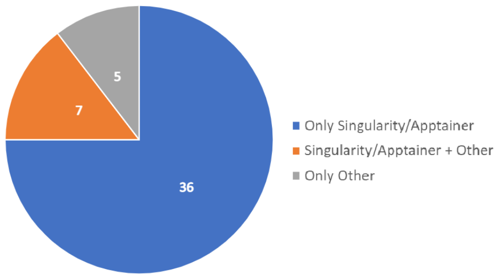

Figure 1. Containerization technologies supported by 48 Top500 HPC providers. Note the majority supports exclusively Singularity and/or Apptainer.

The overall structure of this paper from here on is as follows: In Section 2 we describe the details and the methodology of our investigations, first by presenting the employed hardware and OS configurations (Subsection 2.1) and then by showing the codes, tools and methodology on each level (Subsections 2.2, 2.3, 2.4 and 2.5). Section 3 presents our results at these respective levels. Thus Subsection 3.1 is for our results on level L1, subsection 3.2 is for level L2, while subsection 3.3 shows our results on the topmost level L3. On level L1 and L2 we sort our results regarding different computational resources into separate subsections. Finally, in Section 4 we conclude our results and in Section 5 we present our plans for our related future work.

2. Problem formulation and applied methodology

Since the rise of containerization technology, several investigations have been conducted to describe it's pros and cons. According to [1, 4, 6, 10] CPU and memory performance of containers are about the same or slightly worse compared to the performance of the underlying host system. While in most cases these investigations have shown that the use of containers does not have a significant impact on performance (in some cases they can even beat performance measures of bare metal environments) [2, 3], it is also clear that details change as new versions of underlying hardware and software systems are released. For this reason, in this work we compare scientific calculations of three levels of complexity on both bare metal and in containerized environments, from three different aspects (CPU, memory, I/O) to find out whether actual setups (in early 2023) changed the details of these results.

For the first level (L1), we do "micro-benchmarking" by repeatedly running large amounts of simple calculations. On this level we can easily find the roots of performance strengths or weaknesses of Singularity containers or native environments. The other benefit of this level is that it is very easy to scale these tests to study the size-dependence of time overheads.

Our second level of benchmarking (L2) uses well-known benchmark tools [3, 16] (listed in Subsection 2.3). This level gives us a basis of comparison against others' results. In our current work we focus on CPU, memory, and I/O intensive calculations and tools, since results from this are easy to compare to L1.

As our topmost level (L3) we run an existing pipeline of social-sciences-related scientific calculations, intended for HPC. The exact results and purpose of these calculations don't concern us, but the run-times and other performance indicators give us an understanding of how significantly the behavior of real computational scenarios differs from microbenchmarks and well-known benchmarking tools.

Note that an infinite multitude of measurements can be run with any number of results. Our point is not to declare a winner, but to establish that container portability is not at the price of performance degradation.

2.1. Specification of the applied hardware

We chose one of the most modern available laptop computers, and an LTS Linux version as the hardware and software environments for our tests, so we can say that our results to be found are valid for systems of late 2022 or early 2023. For a detailed description see Table 1 and Table 2.

2.2. About the used codes at level L1

The primary purpose of L1 measurements is to execute simple, often elementary code, repeatedly, targeting a specific piece of hardware or part of it (e.g., an on-processor cache) to measure performance. It is important to note that we cannot speak of a pure processor-memory-storage or video card comparison, because usually a given resource cannot work without the others, but still the main resource

Table 1. Specification of the used hardware.
Date of measurements: 2022–2023.

Hardware	Type / Specifications
CPU	Intel i7-12700H, 14 core, 20 threads, 5 GHz turbo
Memory	32 GB, DDR5, 4800 MHz
Storage	500 GB, Samsung 980 PRO, 7 GB/s, PCIe 4.0

Table 2. Specification of the used software.
Date of measurements: 2022–2023.

Software / property	Native (host)	Container
Userspace	22.10 UwUntu	22.04 Ubuntu
Kernel version	5.15.0.60-generic	—
File system	ext4, ext3 image	can use host's ext4, ext3 overlay
Free space	ext4 host: ~200GB	ext3 overlay: ~70GB
gcc / g++ version	Ubuntu 11.3.0	Ubuntu 11.3.0

can be identified. In addition to the test codes, there are other processes running on the system, and the operating system (although very well optimized), is still running in the background. Because of this, all measurements were performed several times to reduce the chance of measurement error.

The majority of the code is presented without optimization. When optimized, the run time for computation- and memory-intensive processes is significantly reduced in both native and container contexts. Note also that both the container and the native system used exactly the same kernel and compiler. In an HPC environment, it is not common to update these to the current release, so our code may not work with the software installed on the HPC system, or may not compile, or run much slower without certain switches. But, even if not optimized, computationally intensive code will still run significantly faster with a newer gcc or g++ compiler. It is worth using containers precisely because these problems can be completely eliminated, and CPU time is our real time and money.

2.3. Used benchmark tools at L2

In order to have results that are comparable to the findings of other works, we used existing benchmarking tools to test the performance of the native and containerized jobs. We refer to these tools as the standard benchmark tools. The applied so-called standard benchmark tools at L2 are the following:

CPU benchmarks: 7-Zip Compression 22.01 [4, 9, 11, 12]; XZ compression 5.2.4 [11]; CppPerformanceBenchmarks 9; LAME MP3 Encoding 3.100 [11]; FFTW 3.3.6; Geekbench 5.4.6 [7]; Glibc Benchmarks; Himeno Benchmark 3.0; Timed MAFFT Alignment 7.471 [11]; ACES DGEMM 1.0; libjpeg-turbo tjbench 2.1.0

Memory benchmarks: CacheBench [11]; MBW 2018-09-08 [9]; RAMspeed SMP 3.5.0 [9, 11, 12]; STREAM 2013-01-17 [11]; sysbench 1.0.20 [1]; Tinymembench 2018-05-28 [9]

Storage I/O benchmarks: Bonnie++ 2.00 [10]; fio 3.28 [5]; FS-Mark 3.3 [6]; IOzone 3.465 [4, 11, 13]; PostMark 1.51 [9, 14]; sysbench 1.0.20 [1]

Most of these benchmarks were run by the Phoronix Test Suite [11] (PTS), with the exceptions of: Geekbench; sysbench; Bonnie++; and fio. PTS ran each benchmark at least three times, and it's showed result is the average of these results.

2.4. About the used storage I/O benchmarking methodology

Files can be stored in different ways both in native and container environments. We examined three storage options for native environments:

1. using the native (host) filesystem (ext4);
2. using an ext3 image on the host filesystem;
3. using `/dev/shm`.

We found that in Singularity container environments, standard benchmark tools for storage I/O can only be run with one of the following four filesystem options:

1. using the native (host) filesystem (ext4);
2. using an external ext3 image overlay;
3. using an ext3 overlay embedded in the SIF container image;
4. using `/dev/shm`.

`/dev/shm` is a tmpfs memory-backed filesystem, whose maximum size is usually the half of the overall memory size.

For all storage I/O benchmarks there was enough free space on storage to ensure that no significant performance degradation would occur due to insufficient free space being available.

The above options are not all the possible options, however, some were left out, e.g., directory overlay, or tmpfs overlay, due to technical requirements which may not be satisfied on HPC systems, such as needing root permissions. We also didn't examine sandbox-type containers because sandboxes are intended for development purposes.

2.5. About the used pipeline code at L3

For the top level of our investigation, we used a real pipeline of network science C codes [8, 17]. The abstract structure of this pipeline is presented in Figure 2. While the exact purpose of the code is not important, in our case the significance is that load is applied to all of the resources of interest. For example, generating and "attacking" the networks is a highly CPU and memory intensive task, and between the pipeline stages, the network states are serialized to- and deserialized from storage.

Our example pipeline is just one possible scenario from those that were applied in the referred research [8, 17]. We found it important to do these investigations beside the other two levels in order to have a look how the somewhat artificial benchmarking results are related to real-world applications.

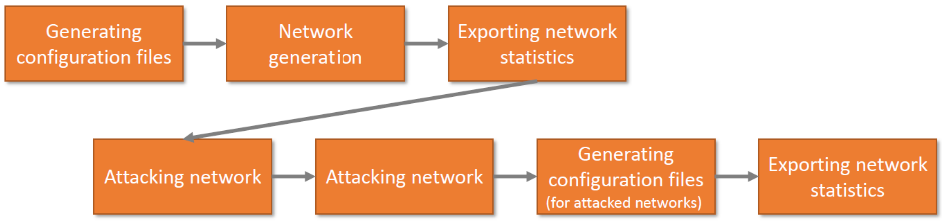
Generating configuration files
Network generation
Exporting network statistics
Attacking network
Attacking network
Generating configuration files (for attacked networks)
Exporting network statistics

Figure 2. The example social network analysis pipeline used to test the performance of native and containerized jobs [8, 17].

3. Results

3.1. Custom microbenchmarking (L1)

The results of our investigations on the lowermost level L1 are summed up in Table 3. In the following subsections we consider the three main system resources and interpret the corresponding findings for both environments as presented in the table. While at first sight it is clear that both the native and containerized environments have strengths, it also has to be noted that the differences are small.

3.1.1. CPU intensive benchmarks (L1)

There are several measurements of a processor's performance because there are many ways to measure its speed, and the result depends on the type of measurement. The measurements therefore are split into the following subcategories:

Processor threads: Single-threaded operation / Multi-threaded operation (with dynamic and static allocation!).
Data structure perspective: Computing with primitives / We use complex data structures.
Cache aspects: Level 1, 2 and 3.
Optimization: compiler flag O2, O3 / O3+ or Ofast / without optimization.

We mainly focus on binaries running on one thread without optimization. The container performed better with operations on whole numbers. No difference was observed for simpler data structures such as char arrays (strings). A Dijkstra pathfinding algorithm was also created from our own implementation. It mostly used the level 3 processor cache for data storage, and did not use non-integer numbers. The native operating system seemed a bit faster in this case.

Table 3. Benchmarking results on level L1. Colored cells mark more than 1% difference.

Microbenchmarks: (Elapsed Time)	Native (s)	Container (s)	Efficiency (%)	Main Hardware Component	file name
whole numbers	540,50	527,41	102,48%	CPU	whole.c
string formatting	473,43	472,92	100,11%	CPU, CPU CACHE	strformat.c
sqrt	413,95	433,72	95,44%	CPU	sqrt.c
sin, cos	378,36	374,84	100,94%	CPU	sincos.c
log	421,33	421,53	99,95%	CPU	log.c
matrix-Dijkstra algorithm	7406,86	7418,63	99,84%	CPU, CPU CACHE	dijkstra.cpp
Memory handling (large size)	299,08	298,95	100,05%	RAM	filereadmem.c
file writing (on ext4 host)	105,73	107,03	98,78%	SSD	filewrite.c
file reading (on ext4 host)	26,45	23,80	111,13%	SSD	fileread.c
file copy (on ext4 host)	22,43	22,02	101,86%	SSD	filecopy.sh
file writing (on ext3 overlay)	124,07	130,59	95,01%	SSD	filewrite.c
file reading (on ext3 overlay)	26,75	25,94	103,09%	SSD	fileread.c
file copy (on ext3 overlay)	33,20	22,31	148,79%	SSD	filecopy.sh

The real differences come in with floating point numbers. While one could do an almost infinite number of measurements using various mathematical functions, we preferred calculating square roots, means, sines, natural-based logarithms. In the case of the square roots, the native performed better, while in the other cases the two were considered equal, or the container was slightly better.

We found that an important thing is to increase optimization, because an optimization of at least O3 can reduce a floating point computation by up to a quarter. Note that when we optimize the root mean square, we found a difference of less than 1%. If we used a 2 years older gcc version, the runtime deteriorated by about 20%. (Which is common, especially if you want to run it on another machine or HPC where they do not update the compilers to the latest version.) So it is more important to keep the software in the container as up to date as possible, because that already has a significant impact on performance.

Summing up the results one can say that CPU usage produces results within

5% error margin, which can become an overwhelming advantage for the container if more advanced software is installed for it in terms of operations on one thread at the L1 benchmark level.

3.1.2. Memory benchmarks (L1)

Since DDR5 memory is so performant, even if there is a large amount of it, it is hard to measure with precision, thus large numbers of large memory allocations, reads, copies and free instructions were performed. In practice, a large part of the memory was allocated, overwritten and freed 256 times. The point was that the amount of data should not fit into the cache of the processor, but also should not be too big in order to avoid swapping. The difference was barely 0.05%, in favour of the container. The conclusion is that, even from the worst perspective, the container handles and uses memory as well as the native operating system at the L1 benchmark level.

3.1.3. Storage I/O benchmarks (L1)

Background storage can also be measured in many ways. At the L1 benchmark level, we look at sequential writes, reads and copies. The container can use the host's ext4 file system or its own custom ext3 (block size 4096kB) overlay file with a removable .img extension. This overlay file can also be used by the host by creating a folder and mounting it. An overlay is useful because you can, for example, install programs into it that the container can use at runtime. The container can use both the host and the overlay filesystem at the same time.

Note that the measurements were made using a PCIe 4.0 SSD drive, which is an order of magnitude faster than a regular SATA III M.2 SSD or any HDD. We performed sequential read, write and copy. (Of course, there is no regularity in the file contents.) The file size was roughly 18 GB.

Generally speaking:
Write: the container is about 2–5% slower than the host
Read: the container is about 3–11% faster than the host
Copying: the container is about 0.5% faster than the host

The host, when it wanted to copy in the overlay, was significantly slower than when the container performed the operation on its own file. For the container it took 22.31 seconds to copy the data on its ext3 filesystem while the same took 22.43 seconds for the host to over its ext4 filesystem. This is only 0.5% of difference. The root of this difference is that ext3 filesystem performs worse than on the ext4 and not the fact that we are using a container.

Of course, there are many ways to read/write copy. Multiple files can be copied in parallel, or many small files in succession, or many large files at once, etc. But for us only writing performed worse in the L1 benchmarks for containers. Of all the benchmarks, storage yielded the largest differences, but the differences are still small, not orders of magnitude. There are seven ways to manage files for the container (many of these ways are presented in section 2.4), so you can choose the most appropriate one depending on the intended use-case.

3.1.4. Conclusions (L1)

Summarizing our results on level L1, we found that the advantage of container portability does not come at the cost of sacrificing performance, as most differences were within measurement error. Regarding memory they are identical. Regarding CPU, the host performs slightly better, but the difference is not radical. Compile-time optimization, however, has a huge importance and the use of state-of-the-art software and compilers in the container is recommended. A gcc or g++ that is only 2 years old, for example, will cause a 20% performance degradation for computationally intensive operations.

3.2. Standard microbenchmarking (L2)

On L2 we present our findings obtained through the well-known benchmarking tools presented in Subsection 2.3. Since there is insufficient space to consider all of our data, we excerpt here only the most interesting but still sufficiently detailed portions. The full dataset can be found in the supplementary material of this work.

3.2.1. CPU intensive benchmarks (L2)

The portion of CPU benchmarks where the differences between native and container environment results are less than or equal to 1% is 82.5% (in this context the difference means how proportionally better one was compared to the other). We consider these differences to be within a margin of error, and in 42.42% of the benchmarks the container won. Where the differences between native and container results were greater than 1%, the differences are these in ascending order: 1.33%, 1.33%, 2.08%, 2.17%, 3.43%, 4.54% and 14.33%. It can be seen that these differences are less than 5% except one case which is 14.33%. The benchmarks producing these deviations were rerun twice. These reruns showed less than 1% differences between the environments in most cases, and the used benchmarks did not consistently judge a given environment type as the faster one. The only exception to this being the 7-Zip decompression benchmark, which reliably showed the native environment performing better, but by only 0.68% and 1.25% (the original run was 3.43%).

Thus the conclusion is that all differences in these benchmark results are within a margin of error, and our benchmarks could not significantly differentiate the two environments.

3.2.2. Memory benchmarks (L2)

The portion of memory benchmarks where the differences between native and container results are less than or equal to 1% is 86.36% (in this context the difference means how proportionally better one was compared to the other). These less than or equal to 1% differences are considered within a margin of error, and in 52.63% of the benchmarks the container won. Where the differences between native and container results were greater than 1%, the differences are these in ascending order: 1.17%, 1.21% and 1.45%. It can be seen that these differences are less than 2%. The benchmarks which produce these differences were rerun twice. These reruns showed less than 1% differences in two of three benchmarks, and the used benchmarks did not consistently judge a given environment type as the faster one.

The conclusion is that all differences in the results are within a margin of error, and the portion of container wins was similar to the portion of native wins.

3.2.3. Storage I/O benchmarks (L2)

Based on our measurement results we can conclude the following:

1. Bonnie++ is unreliable for comparing native and container environments.
2. With Fio's asynchronous non-buffered reading, ext3 container overlay (both external and embedded) wins over host ext4 and native mounted ext3 image.
3. With Fio's reading a file backward, ext3 overlay (both container and native) wins over host ext4 filesystem (both container and native).
4. With FS-Mark sync, IOzone and PostMark benchmarks, host ext4 filesystem is similar to or wins over ext3 overlay (both container and native).
5. With sysbench, host ext4 was definitely better than ext3 container overlays.
6. Except reading a file backward, in everything else native host ext4 was far better than native mounted ext3 image.

Since there are many different filesystem options in both native and container environments and each option has its own advantages, further research is needed to determine which native or container filesystem options or a mix of these are the best for the most common use cases, especially for HPC environments.

3.2.4. Conclusions (L2)

During our investigations we ran 25 different tools and got 103 different results (together with all storage runs in fact 447). To get insight on the overall picture of this large amount of results, we have plotted them on Figure 3. On the figure one can see for all tests a percentage value showing whether containerized or native runs performed better. Note that since in some cases a lower measure is better while in other cases the bigger one, this value cannot be simply written as a fraction of the results coming from native and containerized runs. Here we do not take into account the exact type of the benchmarking, our aim is only to see how native and containerized jobs behave typically. On the figure ">100" means that the difference between the results were more than 100% for either one or the other direction, but the exact value has not been plotted here in order to keep the figure informative for smaller results.

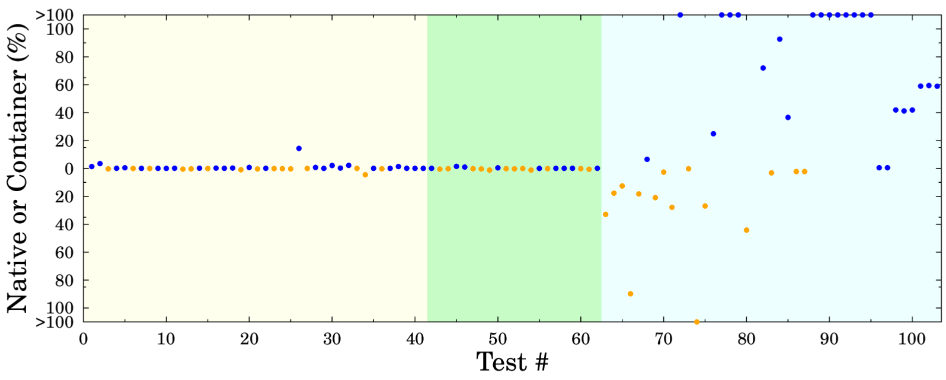

Figure 3. A summary of (L2) benchmarking results. Blue means native wins over containerized runs, orange means the opposite.

Note that in the case of memory related benchmarking we barely found any differences in performance just as for L1. Some CPU related runs, however, resulted in showing that there are cases where native (blue) or containerized (orange) runs are better. The number of cases, however, on each side is almost equal. We found the most diverse results in the case of storage I/O related benchmarking, and here also we see instances on both sides, meaning that it depends highly on the type of benchmark if containerization has a negative or positive effect on the performance. The final message of these investigations should be that one has to know well the required resources of their jobs in order to take the best advantages of containerization – but generally we can say once again that container technology by itself does not require more resources than running natively.

3.3. Benchmarking via a real scientific pipeline (L3)

The used L3 pipeline mostly consists of sequentially running C programs that use each other's inputs and outputs for calculations. These are mainly CPU intensive operations, but they also use memory and storage I/O, e.g., writing out and reading back the results of the steps. Here, along with comparing native and container performance of the unoptimized pipeline binaries, we also compared against optimized binaries, as well as comparing old and new versions of gcc. The results are summed up on Figure 4.

Here, the container has a gcc version released in 2020, while the host has the latest version of 2022. Note that the code ran longer in the older container when not optimized, but the optimization almost completely eliminated the differences (Figure 4 *(left)*). This means that it is worthwhile to install as much up-to-date software as possible in the container to make the runtime sufficiently short and to optimize the code, as this can reduce the runtime of the code by an order of magnitude and almost completely eliminate the differences between the non-matching gcc versions.

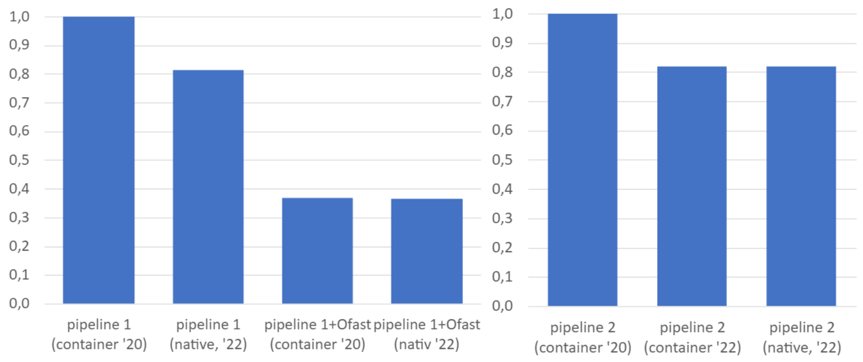

Figure 4. Runtime ratios of a real scientific pipeline. *(left)* shows the obvious profit of optimizing the code during compilation. *(right)* shows how the use of the latest OS can affect the performance.

On Figure 4 *(right)* we ran the pipeline 2 program with the state-of-the-art tools in both cases, but for comparison we also ran it using an older software setup ('20). When we used the old gcc, the runtime increased by 22%. So again the up-to-dateness of the software has more impact on runtime than whether we are looking at a host or native case for L3 Pipeline benchmarks.

4. Conclusions

As the results of our studies we have shown that the use of Singularity containers for providing uniformized, portable environments for scientific calculations does not affect the running times of the calculations in an unaffordably negative way. In some cases the use of containers can even beat the performance of native software environments. We ran our measurements at three levels of complexity. On L1 and L2 we used separate benchmarks for CPU, memory and storage I/O intensive testing. As a result of our findings we have shown with L1 and L2 that there is almost no difference in performance of native and containerized runs of calculations from the aspect of memory intensive tasks. In the case of CPU intensive tasks we found some differences, but most cases showed less than a 5% difference in performance. Storage I/O intensive runs proved to yield the most diverse results, so applications must be careful in choosing solutions appropriate for their workloads.

5. Further work

We have plans for further investigations of the I/O subsytem, and in the direction of parallelization, which we did not study deeply in this work, but is a key component of HPC computations.

Supplementary material

Supplementary material for this work can be found at the below link:
https://arato.inf.unideb.hu/kocsis.gergely/icai2023/

References

[1] S. Abraham, A. K. Paul, R. I. S. Khan, A. R. Butt: *On the Use of Containers in High Performance Computing Environments*, in: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 284–293, doi: 10.1109/CLOUD49709.2020.00048.

[2] C. Arango, R. Dernat, J. Sanabria: *Performance Evaluation of Container-based Virtualization for High Performance Computing Environments*, Microsoft Research WA 98052 (2005).

[3] N. G. Bachiega, P. S. L. Souza, S. M. Bruschi, S. d. R. S. de Souza: *Container-Based Performance Evaluation: A Survey and Challenges*, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 398–403, doi: 10.1109/IC2E.2018.00075.

[4] R. K. Barik, R. K. Lenka, K. R. Rao, D. Ghose: *Performance analysis of virtual machines and containers in cloud computing*, in: 2016 International Conference on Computing, Communication and Automation (ICCCA), 2016, pp. 1204–1210, doi: 10.1109/CCAA.2016.7813925.

[5] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, V. Balakrishnan: *Understanding performance of I/O intensive containerized applications for NVMe SSDs*, in: 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC), 2016, pp. 1–8, doi: 10.1109/PCCC.2016.7820650.

[6] T. A. Bodhanya: *Comparing Cloud Orchestrated Container Platforms : Under the lenses of Performance, Cost, Ease-of-Use, and Reliability*, MA thesis, Uppsala University, Department of Information Technology, 2022, p. 31.

[7] J. Hadley, Y. Elkhatib, G. Blair, U. Roedig: *MultiBox: Lightweight Containers for Vendor-Independent Multi-cloud Deployments*, in: Embracing Global Computing in Emerging Economies, ed. by R. Horne, Cham: Springer International Publishing, 2015, pp. 79–90, isbn: 978-3-319-25043-4.

[8] G. Kocsis, I. Varga: *Investigating the effectiveness of advertising on declining social networks*, Creative Mathematics and Informatics 23.5 (2014).

[9] M. Lindström: *Containers & Virtual machines : A performance, resource & power consumption comparison*, 2022.

[10] P. E. N, F. J. P. Mulerickal, B. Paul, Y. Sastri: *Evaluation of Docker containers based on hardware utilization*, in: 2015 International Conference on Control Communication & Computing India (ICCC), 2015, pp. 697–700, doi: 10.1109/ICCC.2015.7432984.

[11] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, R. Sinnott: *A Performance Comparison of Cloud-Based Container Orchestration Tools*, in: 2019 IEEE International Conference on Big Knowledge (ICBK), 2019, pp. 191–198, doi: 10.1109/ICBK.2019.00033.

[12] A. M. Potdar, N. D G, S. Kengond, M. M. Mulla: *Performance Evaluation of Docker Container and Virtual Machine*, Procedia Computer Science 171 (2020), Third International Conference on Computing and Network Communications (CoCoNet'19), pp. 1419–1428, issn: 1877-0509, doi: 10.1016/j.procs.2020.04.152, url: https://www.sciencedirect.com/science/article/pii/S1877050920311315.

[13] A. Putri, R. Munadi, R. Negara: *Performance analysis of multi services on container Docker, LXC, and LXD*, Bulletin of Electrical Engineering and Informatics 9.5 (2020), pp. 2008–2011, issn: 2302-9285, doi: 10.11591/eei.v9i5.1953, url: https://beei.org/index.php/EEI/article/view/1953.

[14] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson: *Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors*, SIGOPS Oper. Syst. Rev. 41.3 (Mar. 2007), pp. 275–287, issn: 0163-5980, doi: 10.1145/1272998.1273025.

[15] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer: *The 500 most powerful commercially available computer systems*, 2022, url: https://www.top500.org/.

[16] A. Torrez, T. Randles, R. Priedhorsky: *HPC Container Runtimes have Minimal or No Performance Impact*, in: 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), 2019, pp. 37–42, doi: 10.1109/CANOPIE-HPC49598.2019.00010.

[17] I. Varga: *Weighted multiplex network of air transportation*, European Physical Journal B 89.6 (2016).