applied sciences

MDPI

Article

Performance Assessment of Using Docker for Selected MPI Applications in a Parallel Environment Based on Commodity Hardware

Tomasz Kononowicz and Paweł Czarnul *

Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, 11/12 Narutowicza, 80-233 Gdańsk, Poland
* Correspondence: pczarnul@eti.pg.edu.pl; Tel.: +48-58-3471288

Abstract: In the paper, we perform detailed performance analysis of three parallel MPI applications run in a parallel environment based on commodity hardware, using Docker and bare-metal configurations. The testbed applications are representative of the most typical parallel processing paradigms: master–slave, geometric Single Program Multiple Data (SPMD) as well as divide-and-conquer and feature characteristic computational and communication schemes. We perform analysis selecting best configurations considering various optimization flags for the applications and best execution times and speed-ups in terms of the number of nodes and overhead of the virtualized environment. We have concluded that for the configurations giving the shortest execution times the overheads of Docker versus bare-metal for the applications are as follows: 7.59% for master–slave run using 64 processes (number of physical cores), 15.30% for geometric SPMD run using 128 processes (number of logical cores) and 13.29% for divide-and-conquer run using 256 processes. Finally, we compare results obtained using gcc V9 and V7 compiler versions.

Keywords: high performance computing; parallelization; virtualization; Docker; overhead evaluation

Citation: Kononowicz, T.; Czarnul, P. Performance Assessment of Using Docker for Selected MPI Applications in a Parallel Environment Based on Commodity Hardware. *Appl. Sci.* **2022**, *12*, 8305. https://doi.org/10.3390/app12168305

Academic Editor: Sven Gotovac

Received: 31 May 2022
Accepted: 16 August 2022
Published: 19 August 2022

1. Introduction

Nowadays, virtualized environments have gained much popularity thanks to ease of deployment and installation and are being used in various contexts such as: server/cloud deployment of applications provided by various parties possibly taking into account resource preferences of different applications [1]; IoT systems at various levels, i.e., server/cloud [2], middleware [3], gateway/fog [4] and sensor/edge [5] and, finally, in high-performance computing (HPC) systems potentially as a way to deploy applications in the cloud [6]. In the latter, it is of special interest to investigate potential performance impacts as HPC systems have been developed primarily with performance in mind and both execution times as well as scalability measured with speed-ups are of key importance. It can be noticed that virtualized environments can be used to run various workloads in parallel, from, e.g., multiple Cassandra clusters [7], through multiple DeviceHive IoT middleware containers [3] to compute intensive HPC applications [8].

Contribution of this work is investigation of performance and overhead of a virtualized Docker versus native multi-node environment built using commodity hardware (multi-core CPUs and 1 Gbs network) for running parallel MPI applications that are representative of most common parallel programming paradigms, such as: dynamic master–slave (Master-Slave), geometric Single Program Multiple Data (SPMD), as well as divide-and-conquer (DAC) [9]. We perform detailed analysis of execution times and speed-ups for all 1 to n tested numbers of processes and our specific contribution is consideration of best configurations taking into account various compilation flags and numbers of processes which turn out to be different for various applications.

2. Related Work and Motivation

In the literature, authors of several works have addressed the overheads of containerization versus virtual machine solutions.

In paper [10], the authors performed performance comparison of selected high performance computing applications, using bare-metal, Docker (containerization) as well as virtual machines (VM), for various system configurations. Specifically, they tested benchmarks such as High Performance Linpack (HPL) and Graph500 using a system with 2 nodes with Intel Xeon CPU E5- 2670 @ 2.6 GHz and 64 GB of RAM, for a total of 16 logical cores per node with 1Gbps Ethernet. QEMU 2.4.0.1/KVM and Docker 1.7.1 were used. Tested configurations included 2, 4, 8, 16 and 32 instances, each with 16, 8, 4, 2 and 1 vCPUs/execution processes respectively in the case of VM and the same numbers of execution processes in terms of the Docker configurations. Starting with 2 up to 32 instances, performance of the VM configuration for both applications was dropping linearly. For Docker and 2–16 instances, performance stayed at pretty similar levels but resulting in highest performance for 16 instances which the authors attributed to fairer load distribution within each container for a larger number of instances. Additionally, a significant drop in performance for 32 instances was observed for Docker.

In paper [11], the authors compared Docker to KVM for selected benchmarks, using a machine with 2 Intel Xenon E5-2620 v3 CPUs, 64 GB RAM and Ubuntu 16.04. Finding prime numbers resulted in execution time of approx. 19 s vs. 37 s for Docker and VM, respectively, compressing a file using 7-zip gave approx. 8000 MIPS vs. 2000 MIPS, averaged (across add, copy, scale, triad) RAM speed 18,000 MB/s vs. 6000 MB/s, disk read/write time approx. 10 s vs. approx. 25 s, Apache's requests per second 500 vs. almost 200 for Docker and VM, respectively.

The authors of paper [12] argue that for typical HPC environments, several requirements for a container based solution are important: near native performance, compatibility with parallel file systems, workload managers, Docker's images and workflow, as well as security for multi-tenant systems. In this context they mention, as by design more suited for HPC systems than Docker: Singularity [13,14], Charliecloud, Shifter and present their solution—Sarus proving very low overhead on a Piz Daint system showing a difference of only up to 2.7% using 256 nodes compared to a native run for running GROMACS, up to 6.2% using 256 nodes for TensorFlow with Horovod and up to 2.1% for up to 2888 nodes for running COSMO Atmospheric Model Code. In [15], the author presents container solutions, by design more suited for pure HPC environments than Docker, i.e., Singularity, Shifter and UberCloud.

Having said that, Docker is an interesting choice for cloud systems allowing running parallel applications as well as environments in which some of the aforementioned requirements are not crucial. It shall be noted that Docker now supports rootless mode, albeit with restrictions such as ones related to using low numbered ports. Several works focus on comparison of Docker and Singularity as containerization solutions in the context of high-performance computing applications and systems.

In paper [8], the authors investigated Docker versus Singularity performance using up to 64 nodes of a cluster, each one equipped with 16 GB of RAM and an Intel Xeon X3340 running at 2.53 GHz interconnected with 1 Gbps Ethernet. On the software side, benchmarks included NAS-EP from the NAS Parallel Benchmarks, Ondes3D for finite-differences method (FDM) simulation of the propagation of seismic waves in three-dimensional media and a ping-pong MPI application for bandwidth and latency measurements. The authors measured the overheads of Docker and Singularity versus bare-metal and concluded that for 4 nodes/16 ranks and NAS-EP of class B and 1–8 MPI ranks performance of the three solutions was almost identical and for 16 ranks the overhead of both Singularity and Docker was at the level of 9% and 8% respectively. For Ondes3D the overhead of Docker and Singularity is negligible for 1 node/4 ranks, for Singularity increases linearly up to approx. 7% for 16 ranks while for Docker increases dramatically and is at approx. 33% for 8 MPI ranks and approx. 53% for 16 ranks. Execution times for 16 ranks were in the order of 5–10

s for these cases. For a large scale simulation using Ondes3D with times of approx. 125 s and 256 ranks, Singularity overhead was at the level of 1%. The increase of the overhead of Docker was attributed to the overhead of the communication for the overlay network.

In paper [6], the authors provided an assessment of Docker-based configurations and, additionally, Singularity for running several HPC applications. Specifically, they discussed Docker configurations such as:

1. One container per node using the host network and IP of the host node + Infiniband devices of host mapped to containers;
2. One container per node where containers from various nodes are connected through Docker Swarm defined overlay network which enables direct communication between containers. Infiniband devices are available within containers;
3. Multiple containers per host with MPI ranks split among containers—various numbers of ranks per container are possible. A Docker Swarm overlay network is used.

In terms of applications, the authors used the HPCG benchmark, MiniFE (Finite element application), Ohio State University Micro Benchmarks, KMI Hash (K-mer matching interface—measuring integer hashing). The authors compared performance of the various versions and of Singularity versus bare metal. For HPCG and 72 MPI ranks, the best were Singularity, versions 2 and 1 with overheads of 0.22%, 0.47% and 0.65%, respectively. For MiniFE and 96 MPI processes, the best were version 2, version 1 and Singularity with overheads 0.15%, 0.36% and 1.25%, respectively. For OSU and KMI-Hash benchmarks, all versions produced very similar performance. The authors also tested Docker and bare metal versions with Infiniband and Ethernet. For MiniFE and KMI-Hash overhead for Infiniband with Docker versus bare metal was minimal, i.e., smaller than 0.1% while for Ethernet 8.5% for KMI-Hash, 0.4% for MiniFE—version 1. For version 3, running 1 container per rank degraded performance by even 50% but for 2+ ranks per container performance was very close to the other versions.

In paper [16], the authors investigated the performance impact of using Docker for common genomic pipelines and concluded that the overhead can be very low for reasonably long running jobs using an HP BL460c Gen8 system with 12 CPUs Intel Xeon X5670 (2.93 GHz), 96 GB of RAM and Scientific Linux 6.5. Specifically, for a pipeline for RNA-Seq data analysis, they measured the overhead of only 0.1%, for an assembly-based variant calling pipeline, part of a "Minimum Information for Reporting Next Generation Sequencing Genotyping" (MIRING)-compliant genotyping workflow the overhead was 2.4%. Finally, for Piper-NF—a genomic pipeline for detection and mapping of long non-coding RNAs they measured the overhead was much larger—approximately 65% because of many short-lived tasks of median execution time of only 5.5 s.

Apart from comparison of solutions such as Docker and Singularity on bare-metal, in [17], the author considers the context of cloud HPC systems in which containers run in virtual machines. In this context of nested virtualization, Docker is compared to Singularity using several benchmarks. Specifically, tests were performed on an Microsoft Azure platform and VM type was DS1 v2 (General Purpose, 1 vCPU, 3.5 GB RAM, SSDs). Singularity scored better than Docker for the following benchmarks tested: LINPACK (approx. 4% better), IOZone disk read (by approx. 15% for sequential read and 25% for random read) and write (by approx. even 80% for sequential write and by approx. 30% for random write), and netperf compared to Docker overlay configuration (throughput better by approx. 12%, latency by approx. 10%) although Docker host using the underlying network returned virtually the same throughput and latency results. Docker, on the other hand, returned better results for the STREAM memory bandwidth benchmark (by approx. 2.5%).

In [18], the authors compared performance and assessed suitability of the Rkt container technology (providing more security than Docker) for HPC environments and compared it to LXC and Docker when running compute and data intensive applications—HPL and Graph500, respectively. For HPL running on containers Rkt scored very competitively (almost native performance) for larger problem sizes, but was much weaker for smaller sizes for which Docker was better. After analyzing this trend on a single node setup, it was

also confirmed for container clusters. For Graph500 all LXC, Docker and Rkt presented near native performance results.

In paper [19] authors provided a workflow for a container image configuration concerning an all-to-all protein–protein docking application. Container image configurations can be customized using the HPCCM framework and taking into account considering specification differences of HPC environments. The HPCCM framework with the proposed workflow supports both Docker and Singularity and can set container specifications using parameters and proper values, the same approach is used for versions of dependent libraries. Tests were performed on ABCI (1088 nodes with 2 Intel Xeon Gold 6148 and 4 NVIDIA Tesla V100 each) as well as on TSUBAME 3.0 system (540 nodes with 2 Intel Xeon E5-2680 v2 and 4 NVIDIA Tesla P100 each). Singularity, OpenMPI and CUDA were used for protein–protein of all-to-all pairs—52,900 in total. Strong scaling showed 0.964 on ABCI and 0.947 on TSUBAME 3.0 systems, when comparing 64 and 2 nodes. For a larger experiment (1,322,500 pairs) with scaling from 16 to 512 nodes on ABCI strong-scaling was 0.964 and 90 to 180 nodes on TSUBAME 3.0 0.985.

In the context of managing and porting HPC setups and environments, in work [20], the author proposed how to use Docker and Spack together in order to containerize the extreme-scale Scientific Software Development Kit (xSDK) and allow easier management and satisfying required dependencies for HPC environments, especially for non-specialists.

When testing Docker for running an IoT middleware versus a bare metal configuration [3] that might be representative of serving stream of requests from sensors or devices, the authors determined that even though there was an overhead of around 5–10% under high load on the server side (in configurations using either all physical CPU cores or all logical cores on the middleware side), this difference becomes less significant (0–5%) on the client side when communication times are taken into account in the turnaround time. Furthermore, it was shown that using several containers for the middleware helps achieve more throughput at the cost of increased memory usage.

A slightly similar performance investigation of multiple instances of VMWare VMs and Docker containers running concurrently, albeit in the context of multiple instances of a Cassandra database running concurrently was presented in paper [7]. Firstly, for one Cassandra cluster, the authors showed averaged latency of 1.35 ms for Docker vs. 1.9 ms VMWare and 1.2 non-virtualized for replication factor 1, 1.9 ms, 4.4 ms and 2.1 ms for replication factor 2 and 2.2 ms, 5.45 ms and 2.1 ms for replication factor 3 for Docker, VMWare and the non-virtualized environment, respectively. Then, for multiple, i.e., 1, 2 or 4 Cassandra clusters with 3 nodes each, one node on each physical machine. In case of n Cassandra clusters, there were n Cassandra nodes (belonging to different clusters) on each physical machine. The authors reported that for read operations, the maximum number of operations was reduced in case of several concurrent instances, while being increased for write operations in concurrent settings.

Compared to the aforementioned works, we primarily aimed at assessment of virtualized Docker performance in a parallel LAN-based environment built out of commodity components such as desktop CPUs and 1Gbps Ethernet. Such environments are common for: university labs (many are based on several PCs+ Gb Ethernet) and staff/students having access to those, selected companies (same as above using infrastructure for speeding up computations) and even some home environments with 2+ nodes and Gb Ethernet LAN. We shall emphasize that such a cluster built out of commodity nodes (PCs, workstations) is an environment really commonly used in the academic environment and is also feasible for the two other mentioned cases. In line with that, instead of typical aforementioned HPC benchmarks for clusters with top-of-the-line CPUs and fast interconnects like Infiniband, we used parallel codes featuring typical parallel processing paradigms (master–slave, geometric SPMD and divide-and-conquer [9]) that can be used by programmers/users in their applications, not necessarily being typical complex HPC workloads. These could be typical end user workloads that can benefit from parallelization in a multi-node environment with multi-core CPUs that, compared to supercomputing centers, offer some

benefits considering that the user base is quite small, e.g., lack of queuing systems that sometimes require a long waiting phase before computations even start and consequently the possibility to rerun codes with other parameters quickly. We focused on best versions obtained with various compilation flags (-O2, -O3, -O3 -march=native) for all 1 to n tested nodes with assessment of best numbers of processes for particular applications. This is important because these processing paradigms follow various compute/communication schemes and the vast majority of parallel applications can be mapped onto these schemese or a combination thereof. Testing scalability with 2, 4, 8 and 16 nodes allows to assess the speed-up potential of an application and can be used as a first evaluation whether the current implementation can be moved to a much larger cluster (i.e., more nodes and possibly even lower latency interconnect). Using Docker in such an environment allows:

- Easy preparation and testing an application, e.g., in a separate environment and deploying it on such a commodity cluster (e.g., students can prepare at home and deploy at the university). At the same time, Docker allows easy deployment of an application on subsequent computers (with the specific required environment) and scaling with more nodes;
- Easy separation of tasks on such a cluster, e.g., for several research groups to use various parts of the cluster at the same time.

3. Testbed Applications Representing Various Parallel Processing Paradigms

As outlined in [9], the majority of parallel application implementations follow a relatively small number of parallel programming paradigms that can thus be regarded as parallel programming templates. Specific applications fall into these programming paradigms with possibly various parameters, specifically ratios of compute to communication/synchronization times that typically impact speed-ups. In this paper, we test three of such paradigms on one hand, on the other, we test specific application examples implemented with these paradigms. Source codes, implemented with C and MPI, of the latter have been based on examples available as supplemental material to [9] (available at https://static.routledge.com/9781138305953/K35766_eResource.zip, accessed on 2 November 2020).

3.1. Dynamic Master–Slave—Numerical Integration

The concept of this paradigm is to partition input data into independent data packets by a master process, parallelize processing of these packets by slave processes and apply merging by the master process, handling communication using MPI. Partitioning, processing and merging operations are specific to a given application.

In the case of numerical integration tested in the paper, function $f(x) = \frac{1}{1+x}$ is to be integrated over the range $[1, 100]$. The range is partitioned into a number of subranges which correspond to data packets and are initially distributed one per each slave process. Subsequently, each slave process that has finished processing requests a new data packet from the master having sent a result which is immediately integrated into the final result (sum operation on the master part). In general, this scheme implements dynamic load balancing if processing individual data packets takes different amounts of time or these are processed on CPUs of various performance, provided the number of data packets is considerably larger than the number of slave processes (typically a few times larger or more). On the other hand, generating too many data packets decreases the ratio of compute to communication time (due to communication start-up time) and results in poorer performance [9].

Other potential applications following this paradigm might include: numerical integration, block-based matrix multiplication [21], filters applied to a set of images [22], computations of similarity measures of large vectors [23].

3.2. Geometric Single Program Multiple Data—Iterative Stencil Computations

Potential applications implementing this paradigm typically include simulations using iterative stencil loops such as simulations of physical phenomena in 1, 2 or 3D spaces, modeled with differential equations and subsequently solved over successive time steps. Examples include computational fluid dynamics (CFD) codes, weather forecasting [24], heat distribution, Multidimensional Positive Definite Advection Transport Algorithm—part of the EULAG geophysical model [25], medical simulations [26].

Actual implementation tested in this paper corresponds to an iterative stencil loop application in a 3D domain in which a new value for a given cell is computed based on the values of this very cell and neighboring cells from the previous time step. We performed tests with various values of a compute coefficient corresponding to how many computations per cell need to be performed (which may vary depending on the actual application)—one sample is presented in the main results section, more available at the github link presented at the end of the paper. The larger the coefficient, the larger the ratio of computational to communication times and the larger speed-up. Moreover, the implementation optimizes partitioning of the 3D domain into cuboids in such a way that the number of cells assigned to each cuboid, assigned to a different process, is balanced as well as the maximum of the total areas across cuboids is minimized as it corresponds to the number of cells to be exchanged between processes. Additionally, the implementation uses custom MPI data types representing boundary walls: in XY, XZ and YZ dimensions. In each time step, processes update values of the cells of their cuboids and subsequently processes send values of their boundary cells to the processes handling neighboring cuboids. We tested simulation within a 3D domain of size 200 $\times$ 400$\times$ 600 (X $\times$ Y $\times$ Z) for 300 time steps.

3.3. Divide-and-Conquer—Numerical Integration

The implemented divide-and-conquer paradigm follows computations from the root towards leaves of a balanced binary tree and then back towards the root. This scheme starts with a single process and an original and successively generated problems are recursively partitioned into two subproblems. Upon partitioning, rightmost subproblems are passed to another process for parallelization. In the leaves of the tree, custom computations can be performed and results are then merged into a 2-times-smaller number of processes at each level up to the root and the original process. Actual implementation tested in this work sorted function $f(x) = \frac{1}{1+x}$ over range $[100, 200]$ using a parallel merge sort.

Other potential applications following this paradigm might include [9]: recursive searching for a Region of Interest (ROI) within a large image, minimax, $\alpha\beta$ search, etc.

4. Testbed Environments—Physical and Virtualized

The implementation of the experiments required preparation of two environments which were:

- Physical environment (hereinafter referred to as Host);
- Virtual environment using Docker (hereinafter referred to as Docker)—virtualized at the operating system level (containerization).

Due to several optimizations during source code compilation, it was necessary to install the GCC compiler and the OpenMPI library in each of these environments. Computing clusters were created from both environments. The SSH protocol was used for setting up communication between the nodes (used public and private keys). During the experiments, no other calculations were performed within the environments. The test execution times were measured by the `time` program. In order for the tests on the Host and Docker to run in the same hardware configuration, all test runs used the `-hostfile` parameter of the mpirun command, which allowed specific (the same) machines to be used to perform the task.

4.1. Physical Environment—Host

The Host test environment consisted of 17 (identical hardware specifications) physical computers connected to LAN via Ethernet with a speed of 1Gbps. Each of these machines was equipped with an Intel (R) Core (TM) i7-7700 CPU @ 3.60 GHz (4 physical cores, 8 logical threads) and 16 GB of RAM. The operating system installed on these computers was Linux Ubuntu 20.04.3 LTS. For testing purposes, the GCC 9.3.0 compiler version and the OpenMPI 3.1.1 library were installed on all computers in the Host environment.

4.2. Virtual Environment—Docker

To use Docker containerization, firstly, we build an image or use an existing one that contains all the prerequisites needed for the application to work. Docker allows to change the runtime environment and installed programs very conveniently. This allowed the GCC compilers in versions 9.3.0 (OpenMPI 3.1.1) and 7.4.0 (OpenMPI 2.1.1) to be compared with each other. Due to the use of two different versions of the GCC compiler, as in the host environment, two separate images (Dockerfiles) have been prepared.

Created images were based on the very light Linux version for containerization—Alpine. The next step was to download and install the GCC compiler and the OpenMPI library in appropriate versions. It was also necessary to create a new user, configure its settings, and set up SSH communication, which included server installation, use of previously generated encryption keys and port forwarding.

The virtual test environment was built using the same, all physical machines as the Host test environment. The Docker containerization engine has been installed on every computer. Before running the tests, only one container was running on each computer at a time. Each container had access to all host hardware resources (CPU and RAM were not limited in any way). All containers (17) were connected with each other into one computing cluster using the Docker Swarm [27] tool, creating an Overlay network. Two computing containers have never run on a physical host at the same time.

5. Experiments

5.1. Test Methodology and Parameters

The main goal of the experiments was to compare the application execution time on Docker and Host, but in order to make a thorough comparison, many variables need to be taken into consideration. The course of the tests was influenced by:

Program, Paradigm —described in Section 3:

1. DAC;
2. MasterSlave;
3. SPMD.

Compilation parameters, parameters, flags —options used during compilation:

1. No optimization parameter;
2. O2 (-O2);
3. O3 (-O3);
4. O3_native (-O3 -march=native).

Tests without any optimization parameter will not be described in further analysis because such execution times were the longest in these cases (several times longer) and stand out from the rest.

Runtime machine —environment in which the tasks were run:

1. Docker (installed on bare-metal host),
2. Host (bare-metal).

Computing processes —number of processes performing operations: 1–64 with step 4, plus 128 and 256, but only 1, 2, 4, 8, 16, 32, 64, 128, 256 calculation processes will be

presented and 12, 20, 24, 28, 36, 40, 44, 48, 52, 56, 60 calculation processes will be omitted. The available hardware test platform used HyperThreading in order to run 128 and oversubscribe mode to run 256 processes.

The DAC program requires that the number of calculation processes be always equal to a power of two, so it uses from 2^0 to 2^8 calculation processes. The MasterSlave program requires one additional master process that collects results from the slave calculation processes. The number of calculation processes without the master process will always be given in this publication.

The tests were performed as a combination of the above, and each test was repeated 10 times. Before starting the analysis of the results, all tests were completed. Graphs and tables containing median times of program execution in particular configurations are presented in subsequent subsections. Apart from median times, we also present Q1 and Q3 data, interquartile ranges and outliers.

5.2. Results—Execution Times and Speed-Ups

Figures 1–3 show the execution times for the DAC application and a given number of computing processes, using -O2, -O3 and -O3 -march=native compilation flags, respectively. Median values are given in Table 1. The notation in these and remaining figures is standard, i.e., middle dashes represent medians while boxplots represent an interquartile range (IQR) and whiskers max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) respectively, and additionally potential outliers. Since variations in results are small, we have provided detailed plots with selected subranges (using the same scale) in order to better visualize details.

Table 1. Median execution times for DAC.

Compilation	Runtime	Median Execution Time for Computing Processes [s]							
Parameter	**Machine**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
O2	Docker	267.60	137.01	69.10	34.93	18.18	10.22	10.96	10.73
	Host	267.66	137.28	69.18	35.06	18.01	9.56	9.78	9.47
O3	Docker	267.65	137.04	69.13	34.93	18.18	10.23	10.99	10.70
	Host	267.79	137.32	69.21	35.04	18.01	9.58	9.80	9.41
O3_native	Docker	305.70	156.49	78.85	39.81	20.62	11.45	9.58	10.91
	Host	305.91	156.78	78.93	39.92	20.45	10.79	9.86	9.42

Figures 4–6 show the execution times for the MasterSlave application and a given number of computing processes, using -O2, -O3 and -O3 -march=native compilation flags respectively. Median values are given in Table 2.

Figures 7–9 show the execution times for the SPMD application and a given number of computing processes, using -O2, -O3 and -O3 -march=native compilation flags, respectively. Median values are given in Table 3.

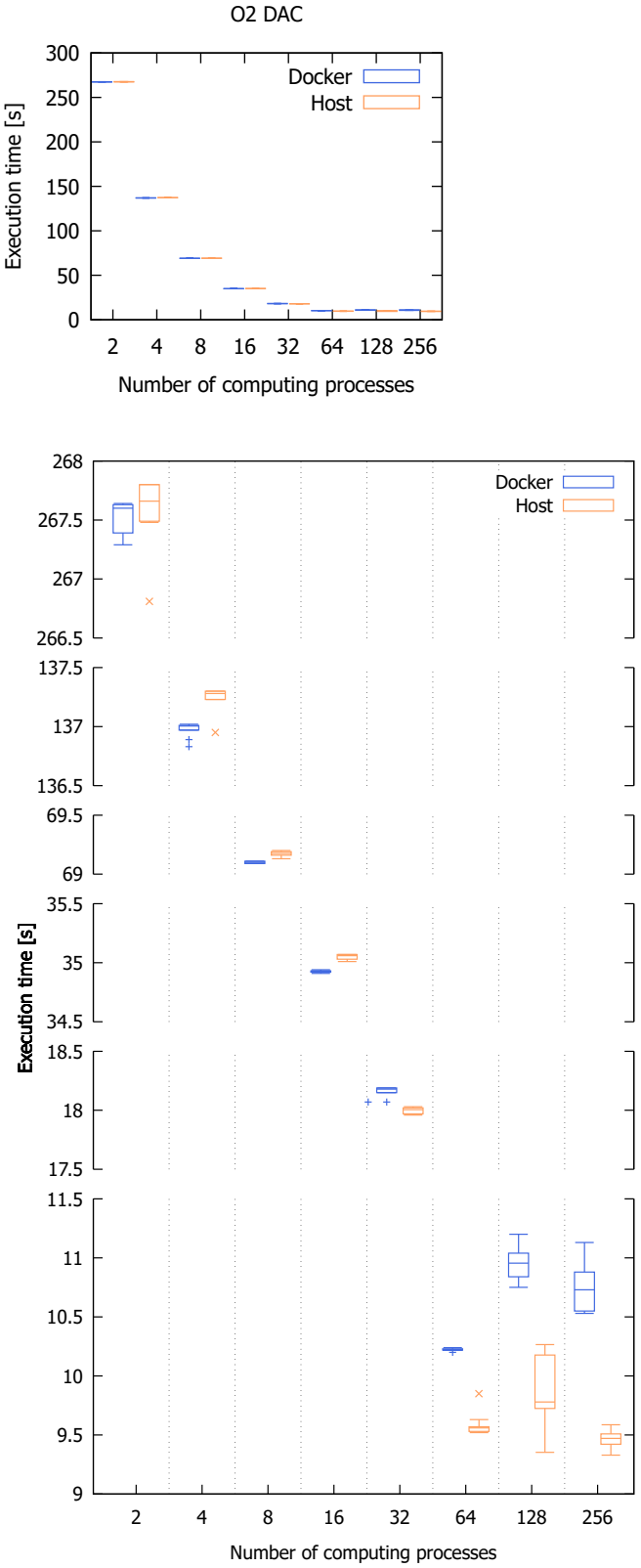

Figure 1. Execution times for DAC, -O2 compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

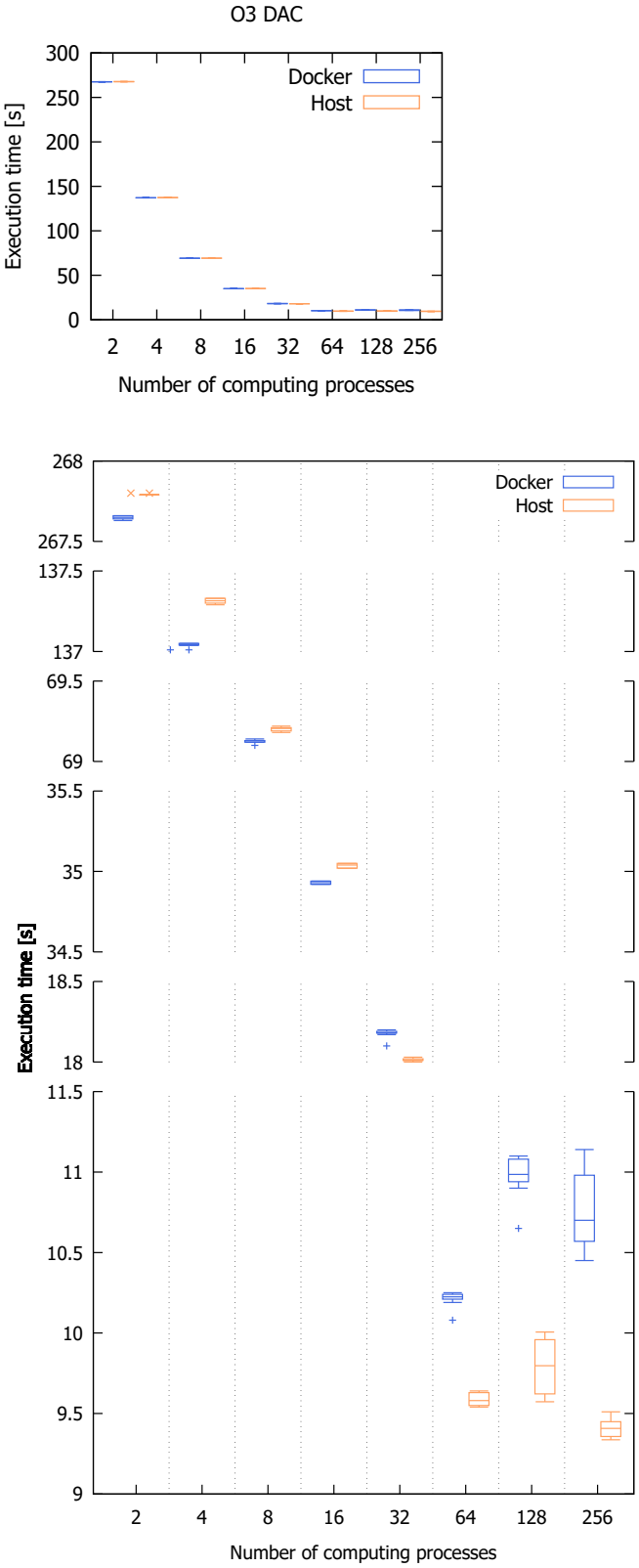

Figure 2. Execution times for DAC, -O3 compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

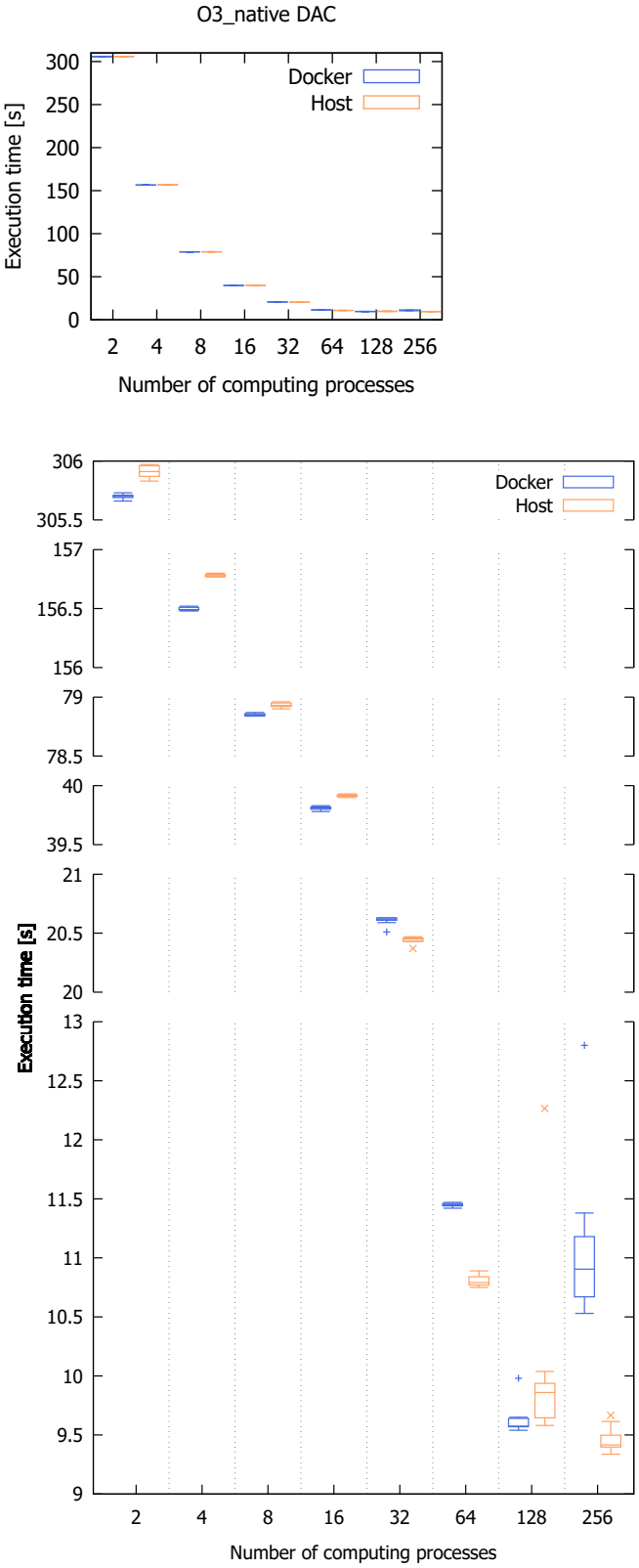

Figure 3. Execution times for DAC, -O3 -march=native compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

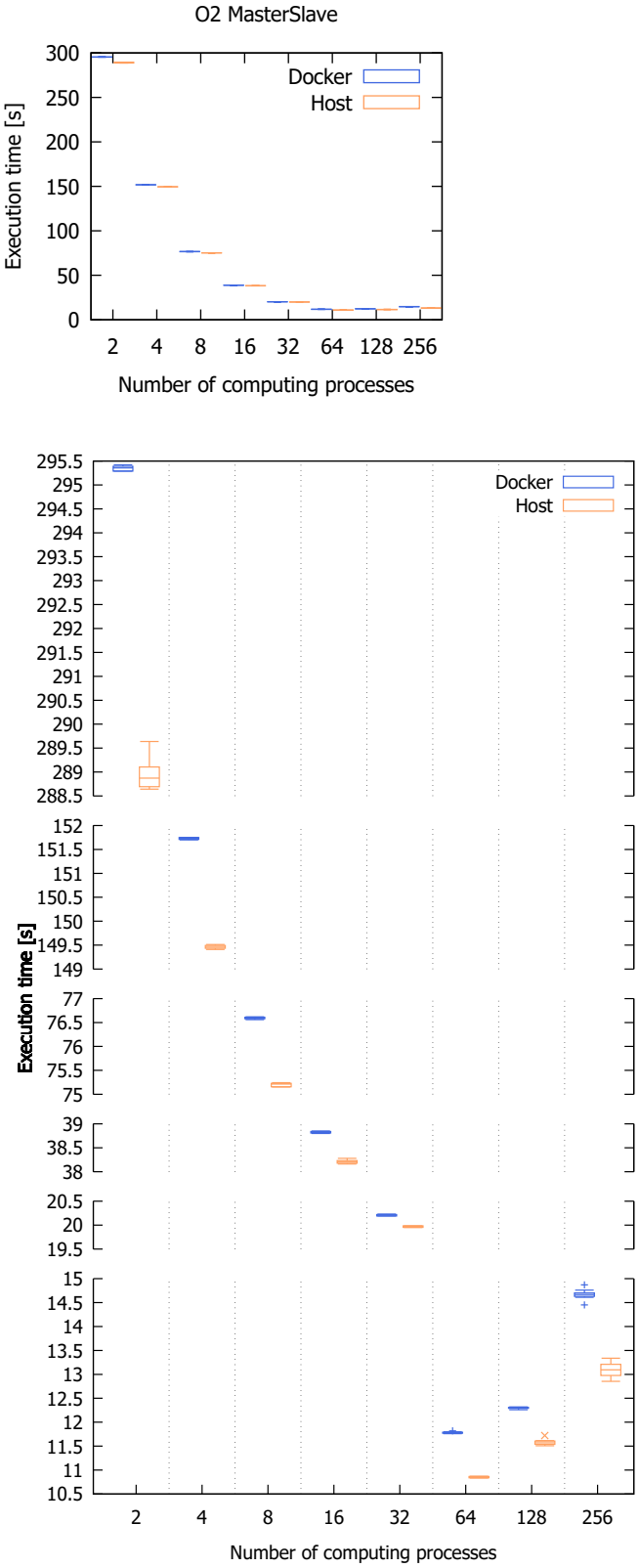

Figure 4. Execution times for MasterSlave, -O2 compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

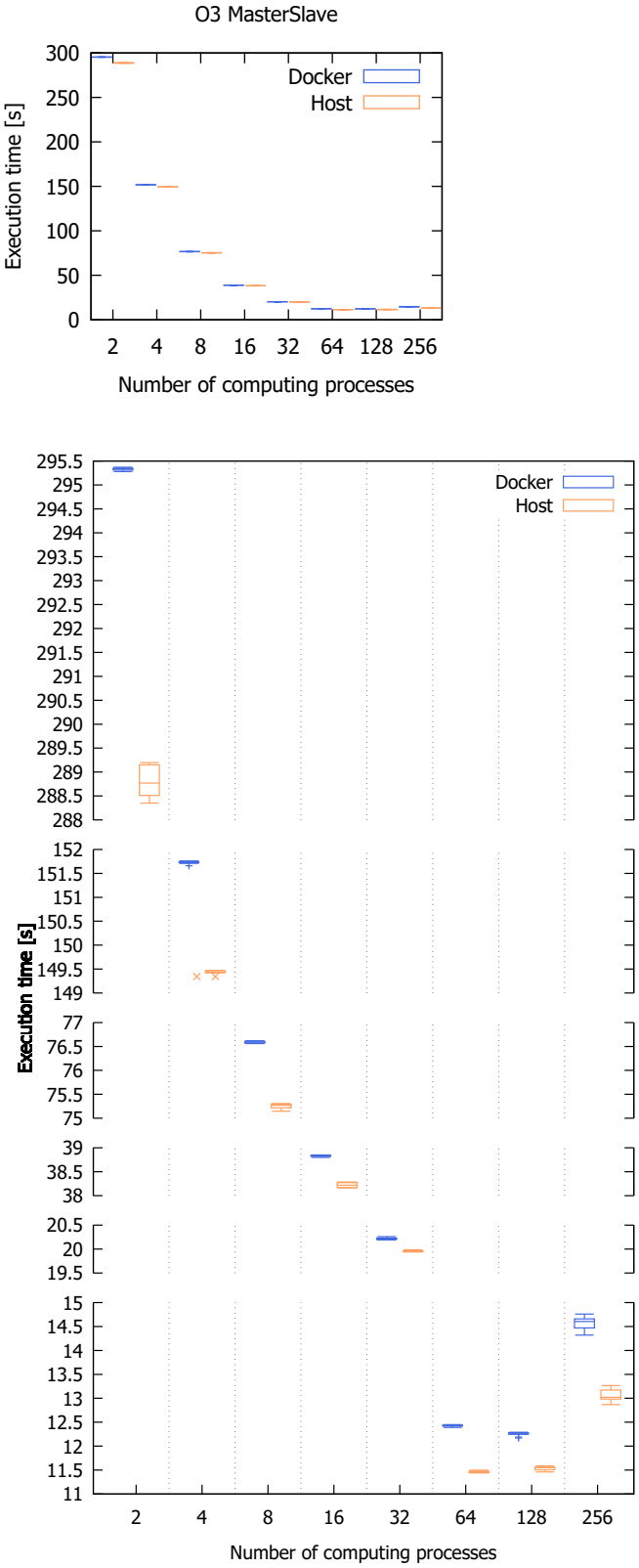

Figure 5. Execution times for MasterSlave, -O3 compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

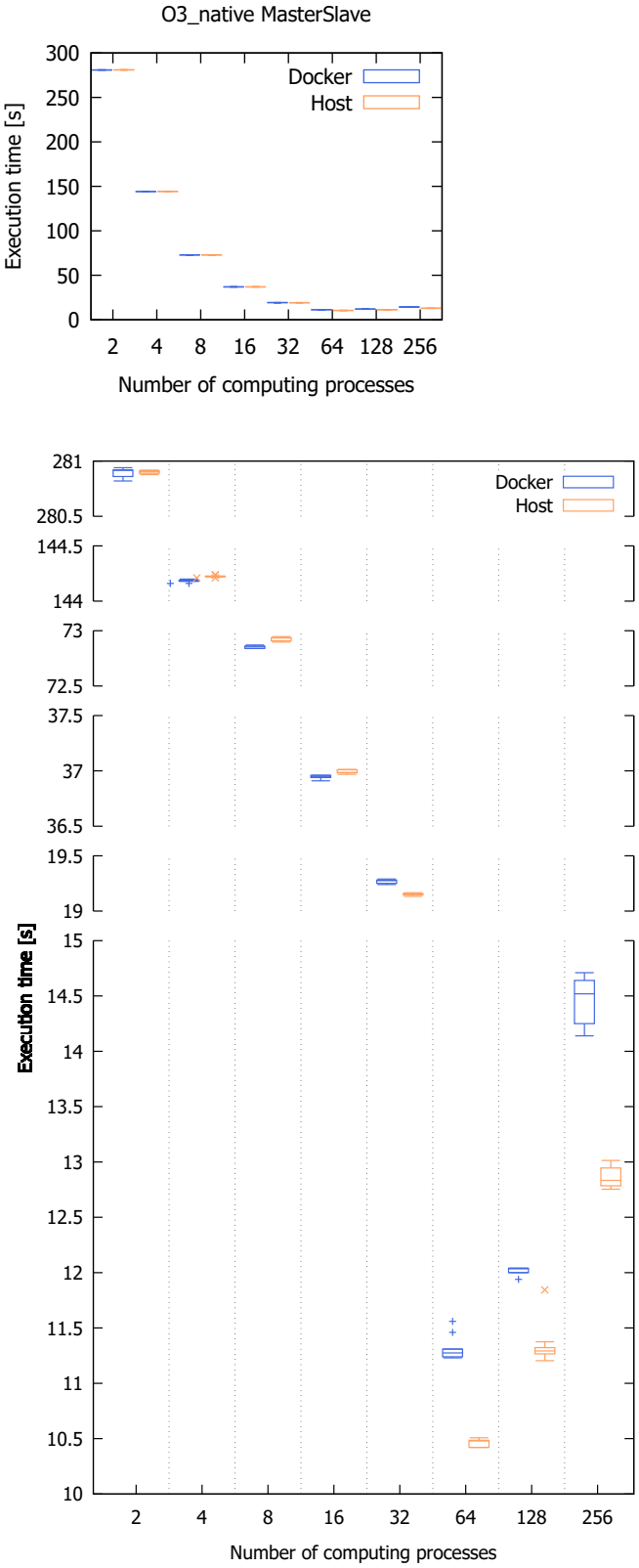

Figure 6. Execution times for MasterSlave, -O3 -march=native compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

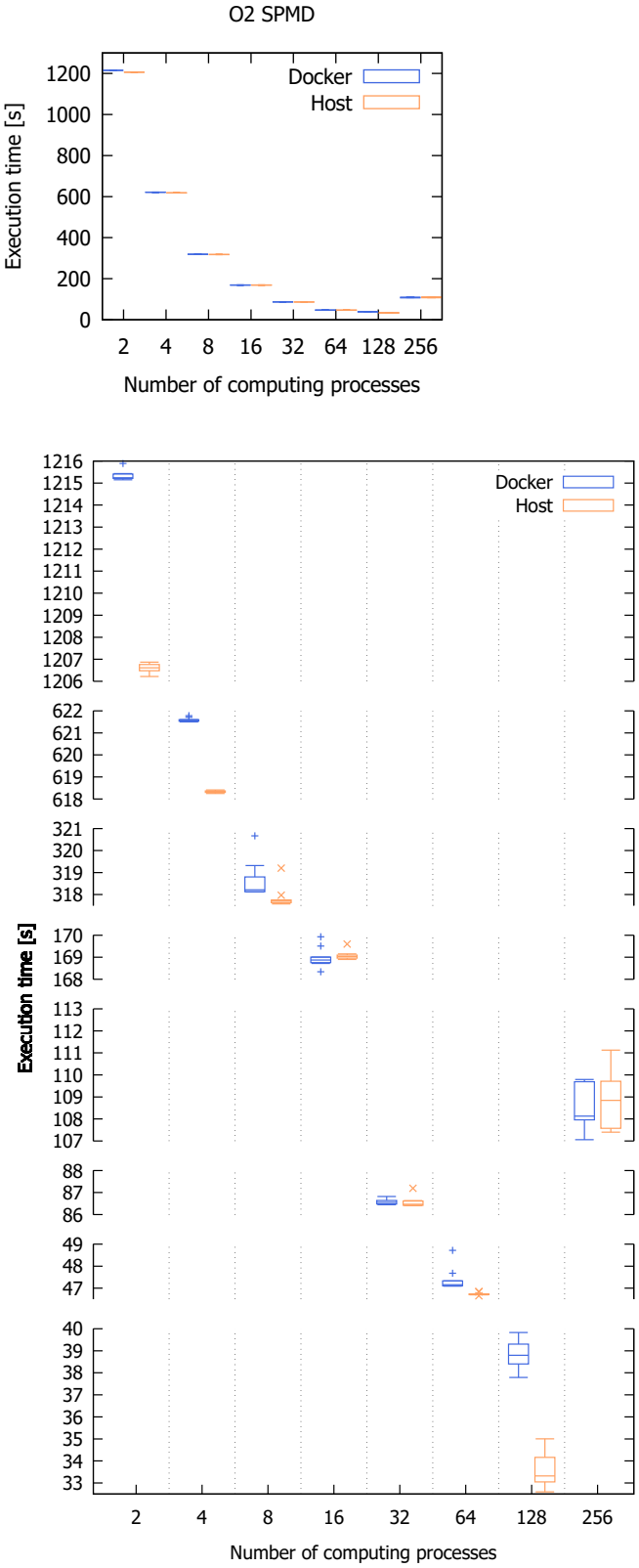

Figure 7. Execution times for SPMD, -O2 compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

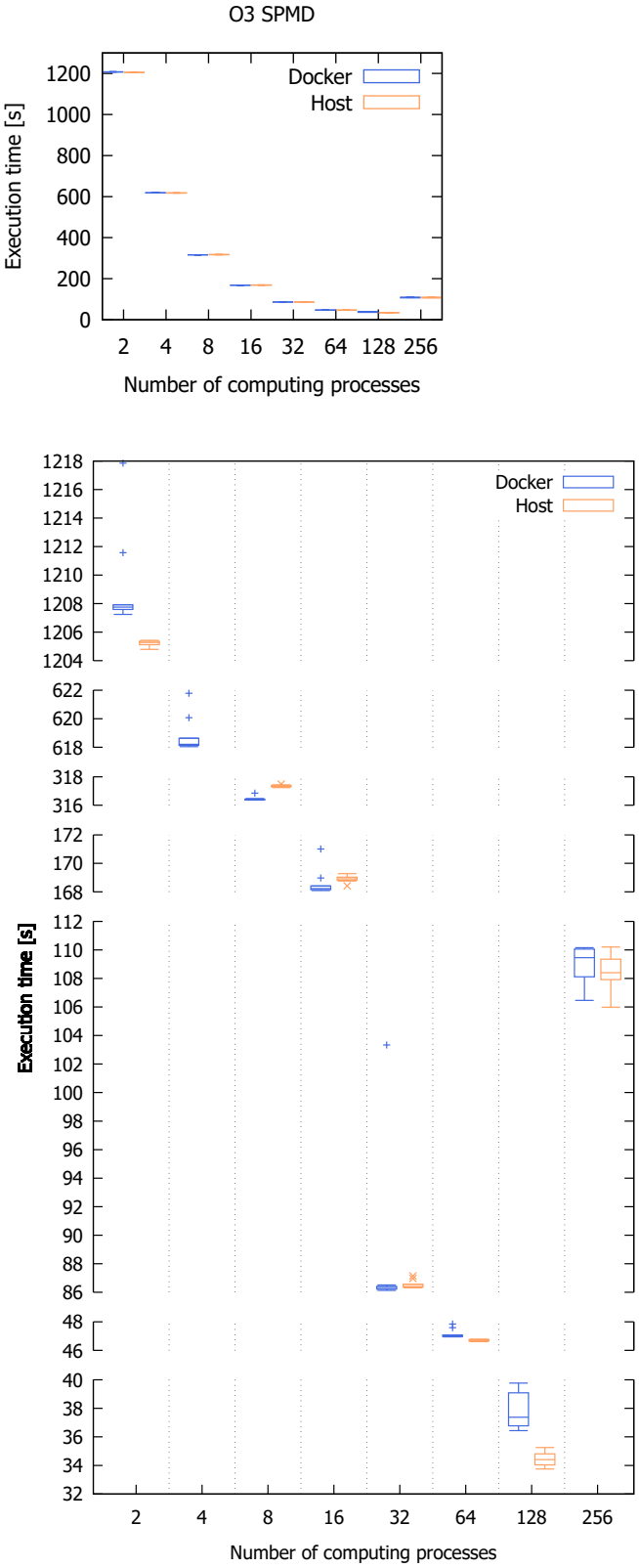

Figure 8. Execution times for SPMD, -O3 compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

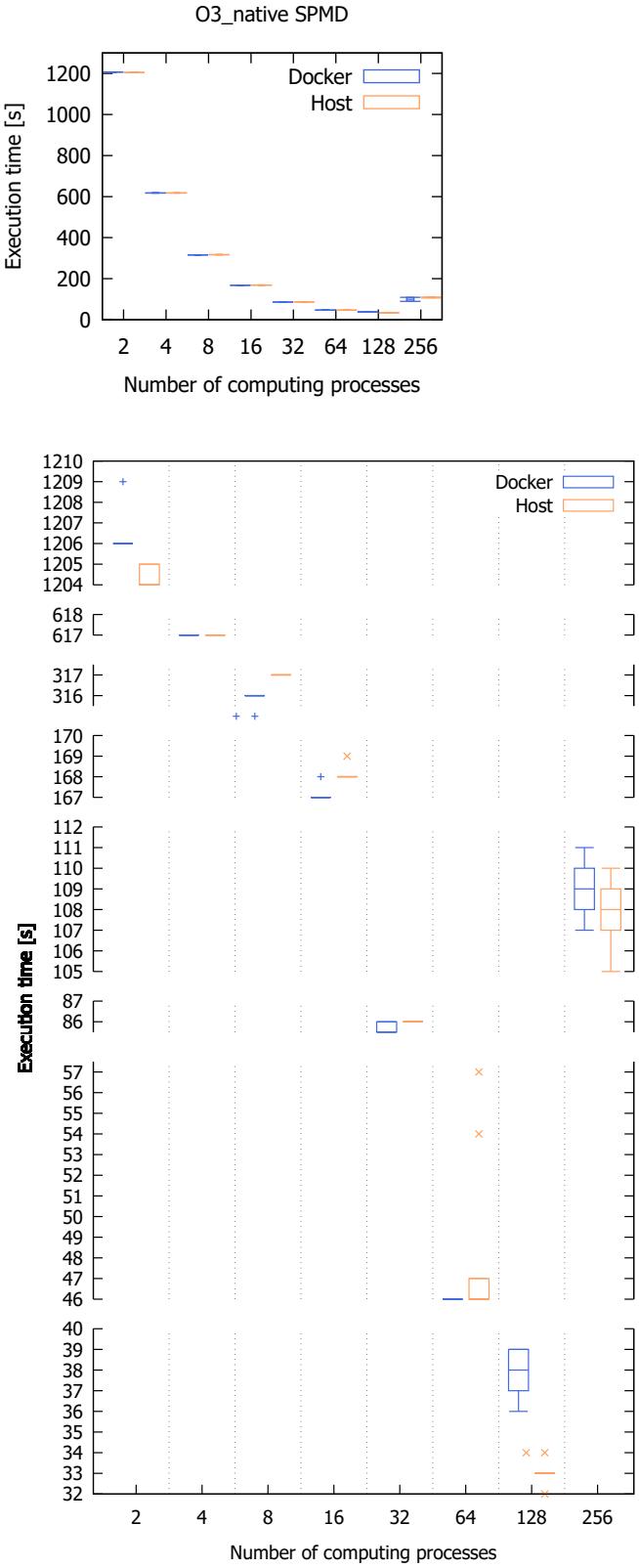

Figure 9. Execution times for SPMD, -O3 -march=native compilation flag; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

Table 2. Median execution times for MasterSlave.

Compilation Parameter	Runtime Machine	Median Execution Time for Computing Processes [s]							
		2	**4**	**8**	**16**	**32**	**64**	**128**	**256**
O2	Docker	295.36	151.74	76.59	38.82	20.22	11.78	12.30	14.67
	Host	288.87	149.46	75.21	38.21	19.97	10.86	11.57	13.09
O3	Docker	295.34	151.74	76.59	38.84	20.21	12.43	12.28	14.61
	Host	288.77	149.43	75.27	38.22	19.95	11.47	11.55	13.02
O3_native	Docker	280.92	144.18	72.86	36.95	19.28	11.28	12.04	14.52
	Host	280.90	144.22	72.93	36.99	19.15	10.48	11.30	12.83

Table 3. Median execution times for SPMD.

Compilation Parameter	Runtime Machine	Median Execution Time for Computing Processes [s]							
		2	**4**	**8**	**16**	**32**	**64**	**128**	**256**
O2	Docker	1215.24	621.56	318.22	168.88	86.56	47.16	38.80	108.13
	Host	1206.62	618.34	317.70	169.03	86.47	46.72	33.32	108.84
O3	Docker	1207.75	618.21	316.42	168.23	86.35	47.00	37.38	109.46
	Host	1205.28	617.70	317.32	168.92	86.40	46.70	34.41	108.41
O3_native	Docker	1206.68	617.46	316.04	167.78	86.04	46.87	38.86	109.30
	Host	1204.94	617.45	317.19	168.80	86.38	46.73	33.70	108.84

Function (1) returns median execution time obtained in all experiments for given arguments.

$$medianExecutionTime(mach, param, prog, proc)[\text{s}] \tag{1}$$

where

mach is runtime machine,
param is compilation parameter,
prog is program,
proc is computing processes.

Parameter names used in this function will be used later in other equations with the same meaning. To simplify getting median execution time for Docker and Host, functions (2) (*d* like Docker) and (3) (*h* like Host) have been introduced:

$$d(param, prog, proc) = medianExecutionTime(Docker, param, prog, proc)[\text{s}] \tag{2}$$

$$h(param, prog, proc) = medianExecutionTime(Host, param, prog, proc)[\text{s}] \tag{3}$$

5.3. Analysis of the Results

Firstly, the fastest version of each program should be chosen, in terms of optimization compilation flags. It should not be assumed a priori which version is the fastest especially because the applications implement various processing paradigms, including various communication schemes. Table 4 presents median execution results for a given parameter, program and number of computing processes. The table presents better median execution times out of Docker and Host versions according to function (4). Figure 10 represents a box plot of execution times for DAC, Figure 11 represents a box plot of execution times for

MasterSlave and Figure 12 represents a box plot of execution times for SPMD, all for better Docker and host versions.

$$median_{dh}(param, prog, proc) = \begin{cases} d(param, prog, proc), & \text{if } d(param, prog, proc) < h(param, prog, proc) \\ h(param, prog, proc), & \text{if } d(param, prog, proc) > h(param, prog, proc) \end{cases} \quad (4)$$

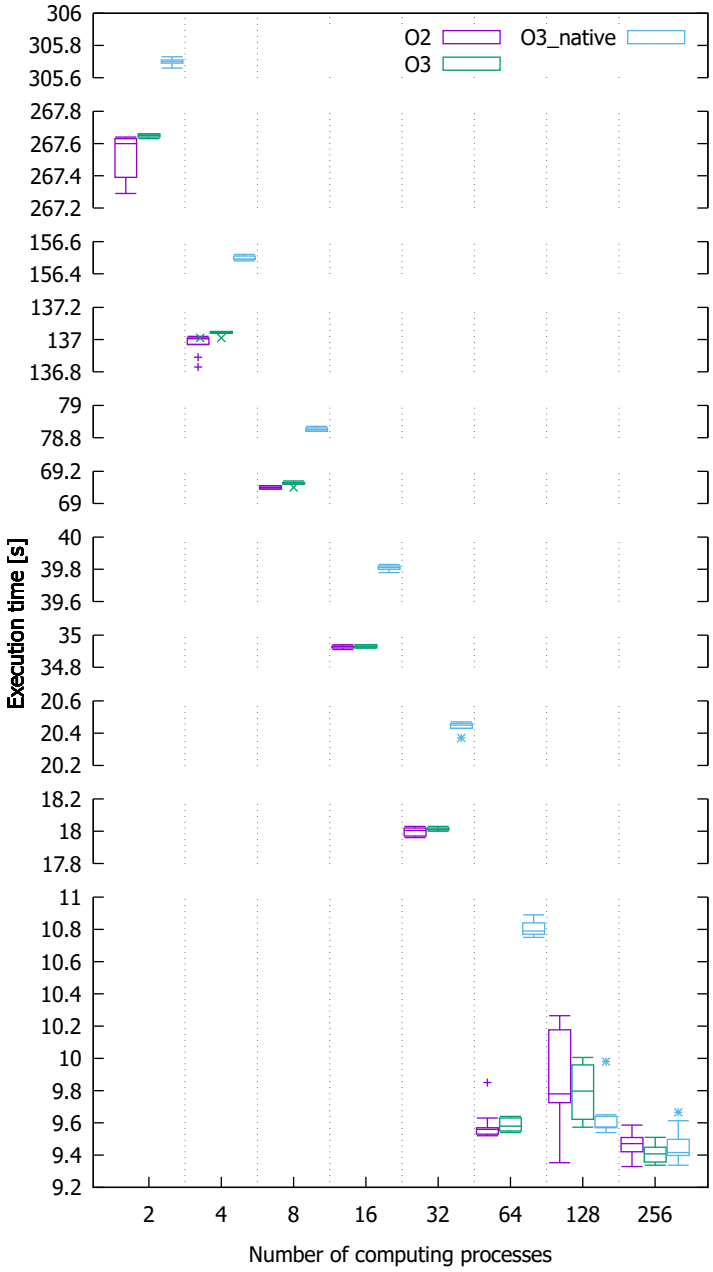

Figure 10. Execution times for DAC, for better of Docker and host versions; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for O2, x for O3, * for O3_native).

Table 4. Median execution time for each parameters, programs and computing processes.

Computing Processes	Program	Median Execution Time [s]		
		O2	**O3**	**O3_native**
1	DAC	522.020	523.520	598.110
	MasterSlave	564.610	564.354	548.795
	SPMD	2361.605	2358.880	2357.340
2	DAC	267.600	267.650	305.700
	MasterSlave	288.874	288.771	280.902
	SPMD	1206.615	1205.275	1204.935
4	DAC	137.005	137.040	156.490
	MasterSlave	149.458	149.430	144.180
	SPMD	618.340	617.695	617.445
8	DAC	69.100	69.130	78.850
	MasterSlave	75.213	75.269	72.860
	SPMD	317.701	316.415	316.040
16	DAC	34.930	34.930	39.810
	MasterSlave	38.211	38.219	36.950
	SPMD	168.875	168.230	167.775
32	DAC	18.005	18.010	20.450
	MasterSlave	19.971	19.946	19.150
	SPMD	86.470	86.345	86.035
64	DAC	9.560	9.580	10.790
	MasterSlave	10.862	11.474	10.480
	SPMD	46.723	46.702	46.732
128	DAC	9.780	9.797	9.575
	MasterSlave	11.570	11.550	11.293
	SPMD	33.322	34.412	33.704
256	DAC	9.472	9.408	9.416
	MasterSlave	13.094	13.022	12.834
	SPMD	108.130	108.406	108.850

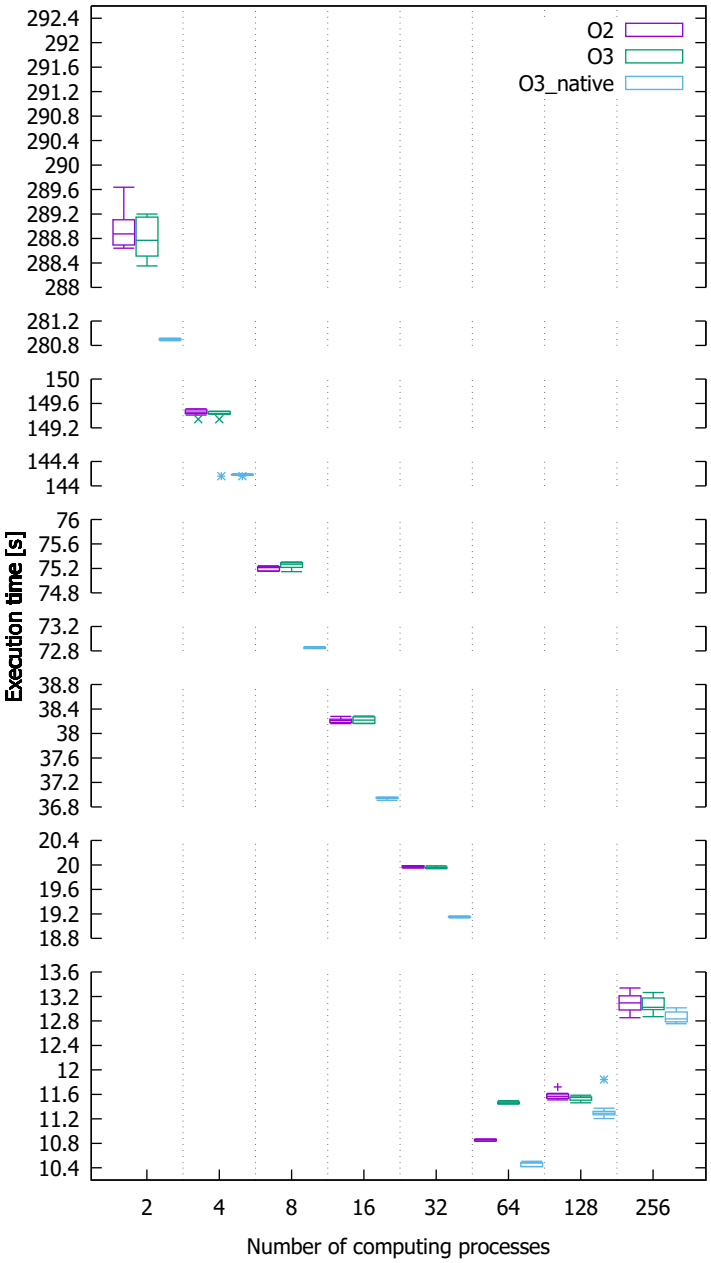

Figure 11. Execution times for MasterSlave, for better of Docker and host versions; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for O2, x for O3, * for O3_native).

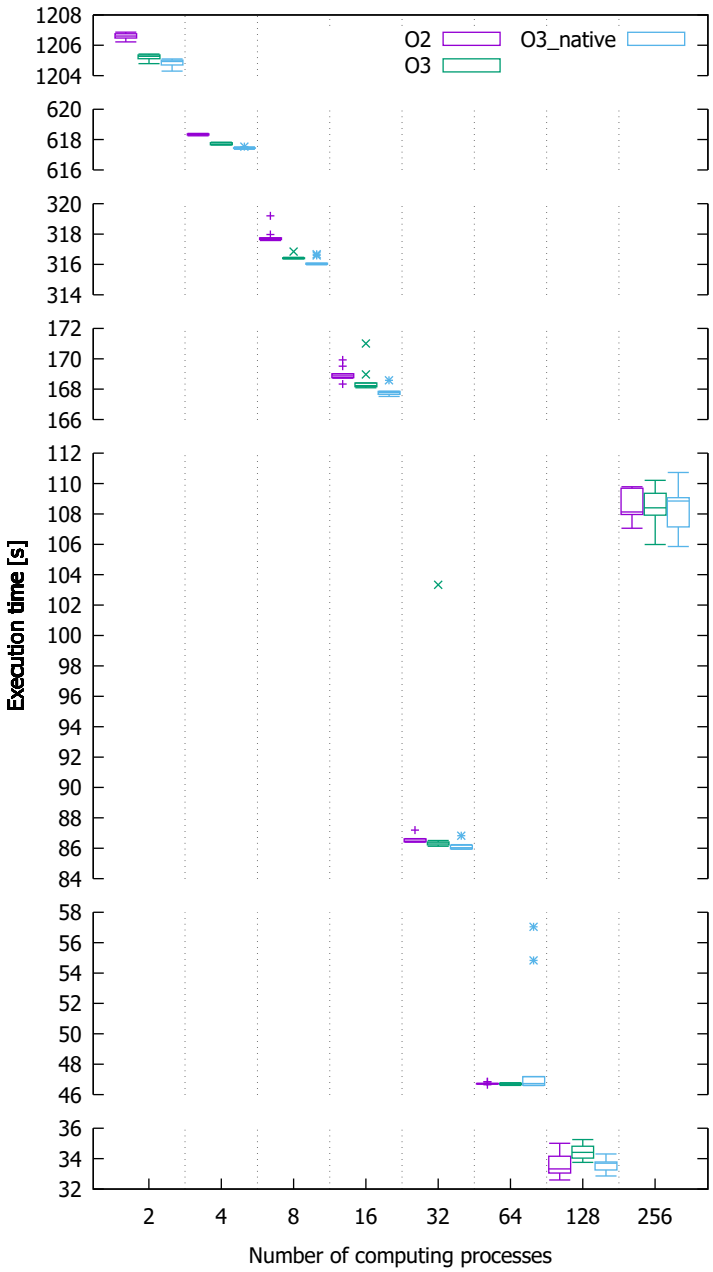

Figure 12. Execution times for SPMD, for better of Docker and host versions; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for O2, x for O3, * for O3_native).

Equation (5) allows to compare two parameters for a specific program. If the result of function p is greater than 1 (the numerator is greater than the denominator), then the program is faster for parameter B. This function sums median execution times for a specific program and given parameter and divides by sum of median execution times for the same program, but with another parameter. Results of function p for various compilation optimization options are presented in Table 5.

$$p(param_A, param_B, prog) = \frac{\sum_{proc=1,2,4,8,16,32,64,128,256} median_{dh}(param_A, prog, proc)}{\sum_{proc=1,2,4,8,16,32,64,128,256} median_{dh}(param_B, prog, proc)} [\text{s}] \quad (5)$$

Table 5. Execution time factor for parameters (results of p function).

Program	$\frac{param_A=O2}{param_B=O3}$	$\frac{param_A=O2}{param_B=O3\_native}$	$\frac{param_A=O3}{param_B=O3\_native}$
DAC	0.9985	0.8756	0.8769
MasterSlave	0.9998	1.0304	1.0306
SPMD	1.0012	1.0020	1.0008

Table 5 makes it easier to infer which parameter should be used in subsequent parts of the experiment:

- For DAC:
  - O2 is faster than O3,
  - O2 is faster than O3_native,
- For MasterSlave:
  - O2 is faster than O3,
  - O3_native is faster than O2,
- For SPMD:
  - O3 is faster than O2,
  - O3_native is faster than O2,
  - O3_native is faster than O3.

Equation (6) is very similar to function p, but due to the different order of operation (function p represents dividing of sums, but function p' represents average of sum of division), the results of the two equations may be slightly different. In our case, both functions allow us to deduce the same best compilation parameter.

$$p'(param_A, param_B, prog) = \frac{\sum_{proc=1,2,4,8,16,32,64,128,256} \left(\frac{min_{dh}(param_A, prog, proc)}{min_{dh}(param_B, prog, proc)} \right)}{9} [\text{s}] \quad (6)$$

For further research, the compilation optimization options were selected as shown in Table 6. We shall also note that the best selected flags are also best when considered individually for either Docker or Host values.

Table 6. The best compilation flag for each program.

Program	The Best Compilation Flag
DAC	O2
MasterSlave	O3_native
SPMD	O3_native

In Figure 13 and Table 7 we compare speed-ups of all programs for Docker and Host with the selected compilation options. Scaling the application for Docker and Host is very similar. We can note that, the best speed-up for DAC and Host is reached using

256 processes (oversubscription) although it is very close to 64 and 128 process configurations, for DAC and Docker it is best for 64 processes; for MasterSlave, using 64 processes; and SPMD—128 processes and falling afterwards. Speed-up was calculated using function (7).

$$speedup(mach, param, prog, proc) = \frac{medianExecutionTime(mach, param, prog, 1)}{medianExecutionTime(mach, param, prog, proc)} \quad (7)$$

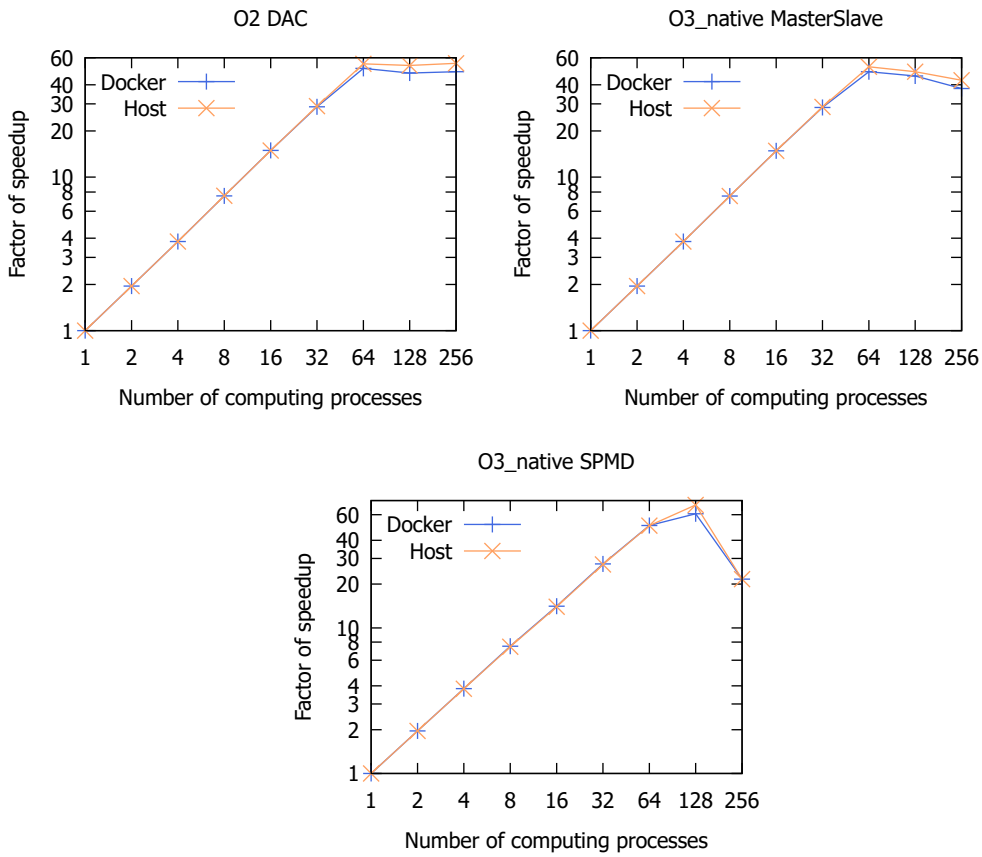

Figure 13. Speed-up execution time of programs for computing processes.

Table 7. Speed-up execution time of programs for computing processes.

Program	Runtime / Machine	Speedup for Computing Processes [s]								
		1	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
DAC	Docker	1.00	1.95	3.81	7.55	14.94	28.71	51.08	47.65	48.65
	Host	1.00	1.95	3.81	7.56	14.91	29.04	54.69	53.47	55.20
MasterSlave	Docker	1.00	1.95	3.81	7.53	14.85	28.47	48.67	45.60	37.80
	Host	1.00	1.95	3.81	7.53	14.84	28.67	52.39	48.62	42.78
SPMD	Docker	1.00	1.96	3.83	7.49	14.11	27.51	50.50	60.91	21.66
	Host	1.00	1.96	3.82	7.43	13.97	27.29	50.44	69.94	21.66

We shall note that we generally expect a decrease in execution time with an increasing number of processes assuming these run on separate physical or logical cores in the system, in our case 64 and 128 respectively, because such a scenario allows shortening the computational phase. This is possible, provided that the communication and synchronization

time do not generate too much overhead that could make the total execution time larger. Generally, for the oversubscription scenario, in our case, 256 processes, we expect additional times due to context switching although in some cases, additional processes (or threads) could hide communication latency.

In the specific cases of our applications, we shall note various communication schemes that affect the execution times and scalability. Speed-up numbers will be specific to particular applications as they depend mainly on the ratio of time spent on computations to the time of communication. For master–slave, this will depend on the subrange size and assumed accuracy of its integration, for geometric SPMD—on the number and complexity of computations performed on a subdomain of a given size, for the divide-and-conquer application—on the size of the initial vector, number of processes and consequently size of a subvector assigned to a process in the leaf of the divide-and-conquer tree. On the other hand, execution times and speed-ups are affected by the communication schemes in these application paradigms and the possibility of parallel communication between pairs of processes. It should be noted that generally, for a given input data size and increasing number of processes, the ratio of computational time to the communication time decreases because the communication time includes the constant component of start-up time [9].

For the master–slave code, we see the maximum speed-up using physical cores (64) and because of the star master–slaves communication pattern in this case, the master process might become a bottleneck for a larger number of processes. For the stencil computations (geometric SPMD) and the 3D domain, each process communicates with a limited number of neighbors which allows for parallel inter-process communication and obtaining highest speed-up for 128 processes using all logical cores. For divide-and-conquer pairs of processes operating at the same level of the divide-and-conquer tree, they can operate in parallel. Here, we see similar speed-up values for 64, 128 and 256 processes. We should note that an MPI implementation can optimize communication between processes running on one node using shared memory, especially for divide-and-conquer and neighboring processes on the same node that explains observed results.

Additionally, we observe that for all the applications and highest obtained speed-ups we see a consistent difference in favor of the host versus Docker versions, i.e., 3.6 for divide-and-conquer, 3.7 for master–slave and 9 for geometric SPMD.

Now, considering the preferred compilation flags, run times for each program on Docker and on the Host can be compared. The results are shown in Table 8.

Table 8. Host and Docker comparison of median execution times.

(CP)	DAC				MasterSlave				SPMD			
	(A)	(B)	(C)	(S_f)	(A)	(B)	(C)	(S_f)	(A)	(B)	(C)	(S_f)
1	0.85	D	0.16	1.0000	0.19	D	0.04	0.9996	9.71	H	0.41	1.0041
2	0.06	D	0.02	0.9986	0.01	H	0.00	1.0000	1.74	H	0.14	1.0014
4	0.28	D	0.20	1.0004	0.04	D	0.03	0.9997	0.01	H	0.00	1.0000
8	0.08	D	0.11	0.9995	0.07	D	0.10	0.9990	1.15	D	0.36	0.9964
16	0.13	D	0.37	1.0021	0.04	D	0.10	0.9990	1.03	D	0.61	0.9939
32	0.17	H	0.97	0.9888	0.13	H	0.65	1.0065	0.34	D	0.40	0.9960
64	0.66	H	6.90	0.9339	0.80	H	**7.59**	1.0759	0.14	H	0.30	1.0030
128	1.18	H	12.02	0.8913	0.74	H	6.58	1.0658	5.16	H	**15.30**	1.1530
256	1.26	H	**13.29**	0.8813	1.69	H	13.14	1.1314	0.45	H	0.41	1.0041

(CP) Computing processes

Argument *param* in functions (2) and (3) (d and h) means the best compilation parameter for program (from Table 6). It is used to explain values in columns **(A)**, **(B)**, **(C)** and **(S_f)**.

(A) Difference between execution time on Docker and Host with the selected compilation flag

$$a(param, prog, proc) = |d(param, prog, proc) - h(param, prog, proc)|[\text{s}] \quad (8)$$

(B) Faster runtime machine (D—Docker, H—Host).

$$b(param, prog, proc) = \begin{cases} D, & \text{if } d(param, prog, proc) < h(param, prog, proc) \\ H, & \text{if } d(param, prog, proc) > h(param, prog, proc) \end{cases} \quad (9)$$

(C) Answer to question: "How many percent the Docker is faster (if column (B) equals D) or slower (if column (B) equals H) than Host". For example—DAC program and 32 computing processes: Docker (because H is present in column (B)) is 0.97% slower than Host. Values from this column are presented in Figure 14.

$$c(param, prog, proc) = \left|1 - \frac{d(param, prog, proc)}{h(param, prog, proc)}\right| \times 100\% \quad (10)$$

(S_f) Factor of speed-up indicator for Docker and Host.

$$S_f(param, prog, proc) = \frac{speedup(Docker, param, prog, proc)}{speedup(Host, param, prog, proc)} \quad (11)$$

As we can see for all the applications, up to 32 computing processes, it does not practically matter whether these programs are run on the Host or Docker and differences between machines (**(C)** column) are smaller than 1%. Using 64 computing processes (highlighted) and more, all programs were executing faster on the Host.

For all applications using 64 computing processes and more, the percentage difference (column **(C)**) between Docker and Host for DAC and MasterSlave becomes greater than for a smaller number of processes (except SPMD for 256 computing processes—but this case is already after a significant drop in performance so is not in the area of interest). We put in bold the Docker overhead (in percent) for the number of processes for which the given application obtained the shortest time. For such configurations, we see what we can consider a Docker overhead of between 7.5% and 15.3%.

It can be noted that in [28], the authors concluded that for a artery CFD code case in a Lenox cluster (4 nodes with 2× Intel Xeon E5-2697v3 and 1GbE network) and using Docker for a total of 112 threads, the Docker overhead increased with an increasing number of MPI ranks, i.e., it was close to bare metal for 8 × 14 (MPI ranks × threads per rank) and 16 × 7, but approximately 10% for 28 × 4 and 56 × 2 and even approximately 40% for the 112 × 1 case.

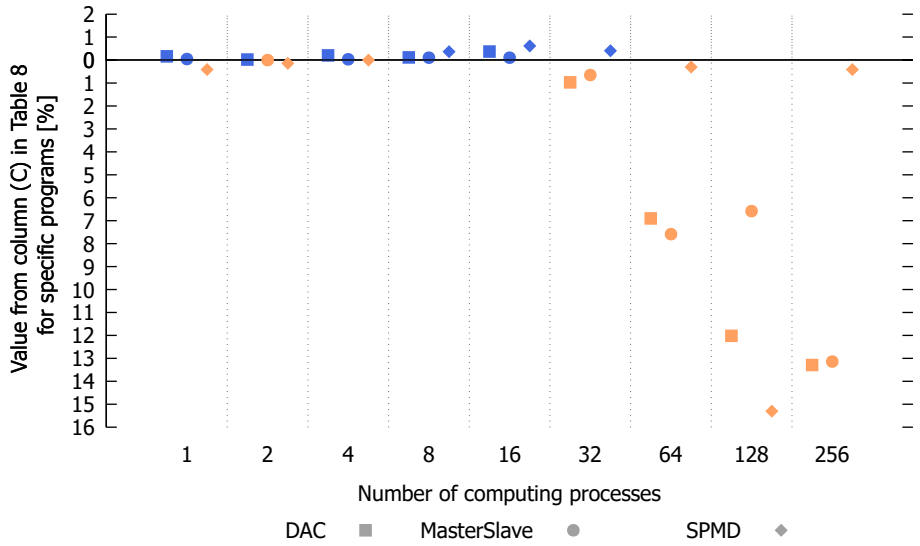

Figure 14. Host and Docker comparison of execution times; orange points denote the overhead of Docker while blue points correspond to minimally better Docker execution times.

Due to the possibility of a very easy change of the runtime environment using Docker (as mentioned in Section 4.2), for the sake of performance evaluation of potential progress regarding compilers and software development, a comparison of the execution times using GCC 7.4.0 with OpenMPI 2.1.1 (hereinafter referred to as V7) and GCC 9.3.0 and OpenMPI 3.1.1 (hereinafter referred to as V9) versions was made. In practical terms, though, we would typically use newer stable versions of the compiler and software rather than older ones since such are theoretically improved and maintained better. Table 9 shows for which version of the compiler the compiled program ran faster, using Docker. If the program compiled by the V7 compiler version was faster than using version V9 (for a given configuration), then a respective cell contains V7, otherwise, it contains V9. The value in parentheses denotes in percent how much faster the faster version of program was. For example, the SPMD program compiled with the V9 version on Docker with 64 computing processes performed 0.67% faster than the SPMD program compiled V7 version with the same configuration (Docker machine with 64 computing processes).

Table 9. Better compiler version using Docker.

Computing	Program		
Processes	DAC	MasterSlave	SPMD
1	V9 (17.26%)	V9 (0.17%)	V9 (0.98%)
2	V9 (17.00%)	V9 (0.15%)	V9 (1.00%)
4	V9 (16.98%)	V9 (0.14%)	V9 (0.90%)
8	V9 (16.85%)	V9 (0.10%)	V9 (1.14%)
16	V9 (16.67%)	V7 (0.15%)	V9 (1.04%)
32	V9 (15.89%)	V7 (0.73%)	V9 (0.94%)
64	V9 (14.26%)	V7 (2.48%)	V9 (0.67%)

Percentage differences of program median execution time between V7 and V9 for MasterSlave and SPMD are very small. For MasterSlave, it is always below one percent except for 64 computing processes (2.48%) and for SPMD, it is always about one percent (0.67–1.14%). For the DAC program (-O2 flag compilation selected), percentage difference between V7 and V9 for 1, 2, 4, 8.16 and 32 computing processes is about 17% in favor of V9.

For 64 processes, the V9 version is approximately 14% faster. When using the -O3 flag for the DAC program, differences between a program compiled with compilers of versions V7 and V9 are similar to those shown in Table 9 for MasterSlave and DAC.

6. Analysis Considering Hardware Counters

We have performed several measurements and extended the paper with results of those and corresponding comments that include measurements of both:

1. Hardware counters referring to the CPU side measured with the perf tool. Selected metrics:
   (a) Cache misses—how often data in cache were unavailable or out of date (prefetch requests included);
   (b) Context switches—number of CPU context switches from one process or task to another;
   (c) Instructions—average number of instructions executed for each clock cycle;
   (d) Task clock—CPU utilization during program execution.
2. Communication performance. For this test, we included results of the MPI communication benchmark measuring point to point performance with contention, i.e., communicating pairs of processes (https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src3/pingpong/C/mass/main.html, accessed on 22 July 2022) which we believe very much corresponds to our cases since in SPMD code pairs of processes of neighboring subdomains exchange data in parallel in respective dimensions, in divide-and-conquer pairs of processes exchange data in successive steps of the divide-and-conquer tree (various numbers at various levels) and in master–slave, we deal with master–slave exchanges (one at a time but MPI can buffer messages from many slaves).

For the hardware counters, we see interesting results with various metrics better either for the Docker or host cases, for the selected compilation flags. For DAC results presented in Figure 15, cache misses and context switches are larger for host, task clock is slightly higher or same for Docker, while instructions per cycle same or larger for host. For the master–slave application and slave (i.e., performing majority of computations), the results shown in Figure 16 cache misses are same for 128 processes, but larger for host up to 64 processes, context switches larger for host, instructions per cycle larger for host for 32–128 processes and task clock slightly lower for host for 32–128 processes. For SPMD results presented in Figure 17, cache misses are very similar, marginally better for Docker, context switches larger for host, instructions per cycle are larger for host for 64–128 processes, task clock very similar for both cases. In general, we can conclude that instructions per cycle are higher for host for larger numbers of processes, while cache misses and context switches either similar or larger for host.

For the communication tests in Figure 18, we present results for various numbers of processes, from 1 to 128 (without oversubscription) for selected data sizes which correspond to our tested applications. Specifically for master–slave and divide-and-conquer—processes exchange data corresponding to start and end of range to be integrated or result—1–2 doubles; for geometric SPMD—processes exchange data of 2D walls with direct neighbors processing neighboring subdomains. In this case, for instance, for 64 processes and domain of $200 \times 400 \times 600$ message sizes are 120–160 KB while for 128 processes and the same domain size message sizes will be 60–120 KB. It should be noted that for the same domain size and a smaller number of processes, sizes of messages will increase and will be, e.g., 640–960 KB for four processes. Following these values, we present measured bandwidths for the following data sizes: 16 B, 16 KB, 32 KB, 64 KB, 128 KB and 1 MB.

We have noted in the measurements that for messages of size smaller than 16–64 KB measured host bandwidths are higher but then for larger message sizes and specifically smaller number of processes Docker configurations exhibit slightly better values, apparently due to internal mechanisms. For 16–64 KB messages and 128 processes bandwidths for host can be better. We shall observe that these trends are in line with our observations, i.e., for SPMD visibly better execution times for host for 128 processes, very similar for 64 and marginally, but visibly worse for smaller (i.e., larger messages) and for master–slave and divide-and-conquer practically same or better results for host (considering measurement errors—slightly better results for Docker are smaller than 0.4% for O2 DAC and O3_native master–slave). We should also note that with an increasing number of processes and same input data size, ratio of communication vs. computational time generally increases and thus larger observed differences for DAC and master–slave, otherwise showing very similar results for smaller numbers of processes.

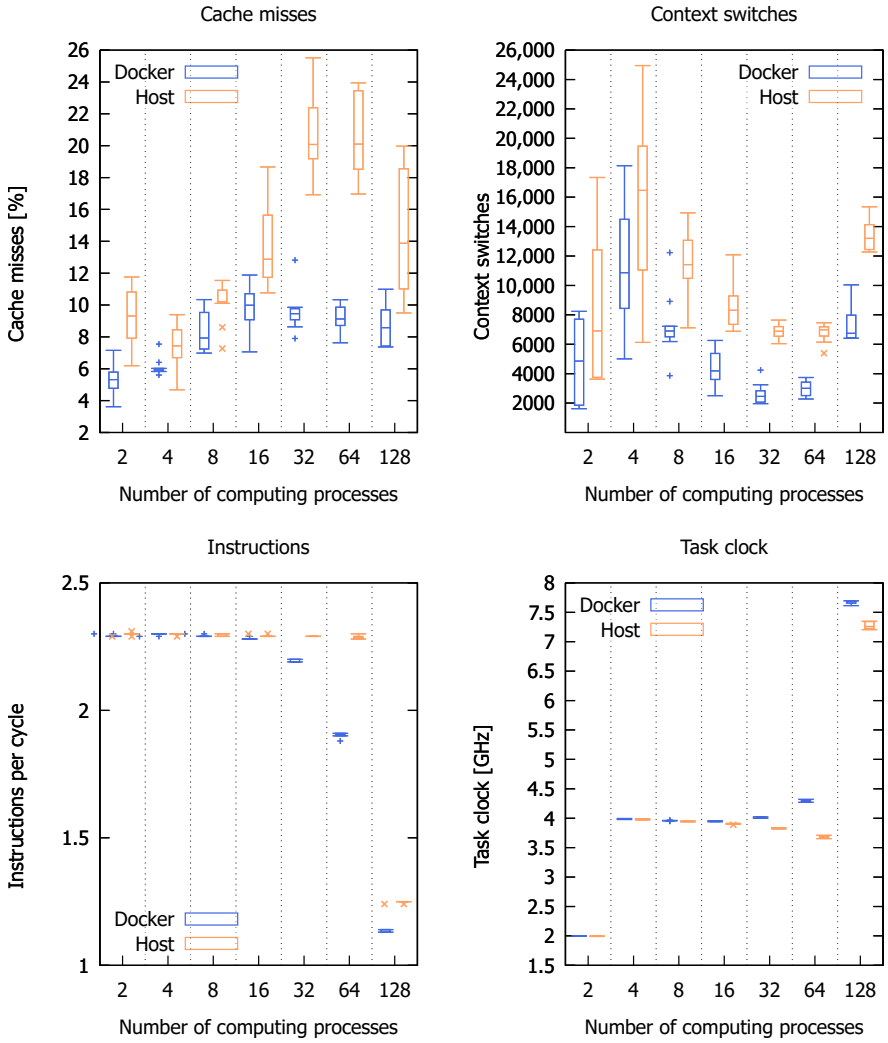

Figure 15. Hardware counters for DAC program with -O2 compilation parameter; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

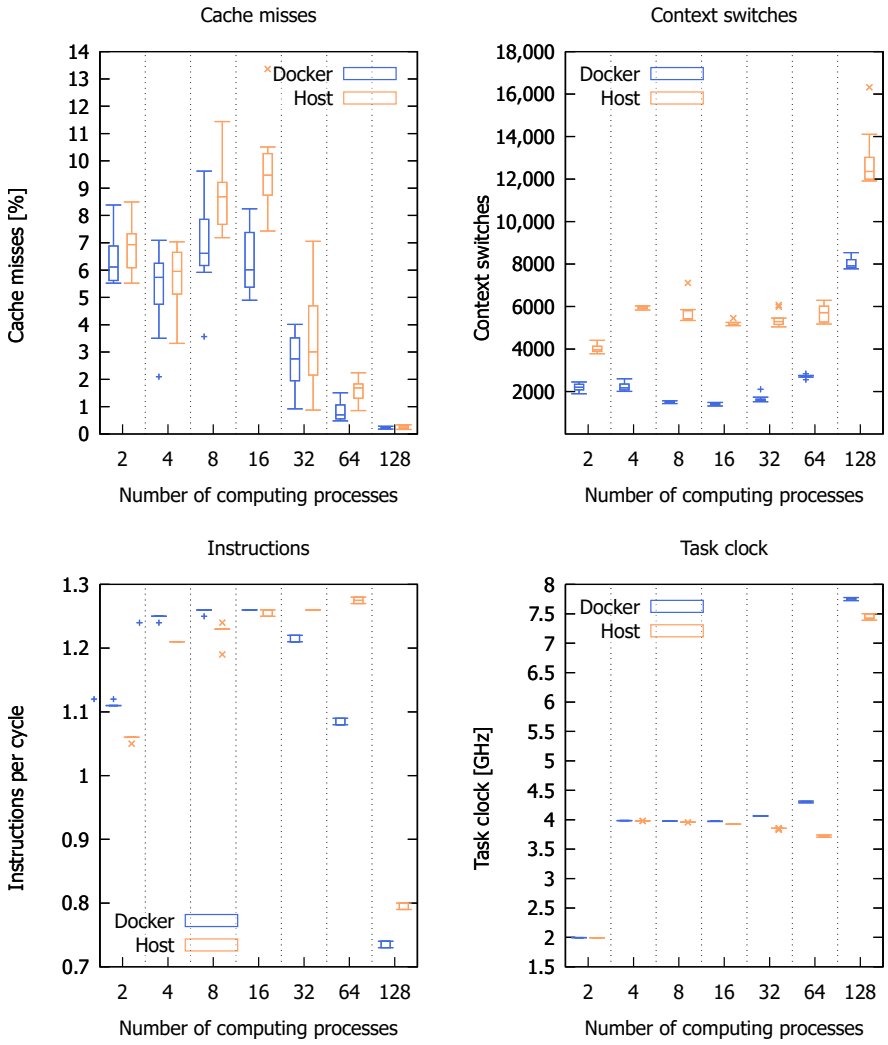

Figure 16. Hardware counters for MasterSlave program with -O3 and -march=native compilation parameters (slave node); boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

Communication latency between all nodes (peer-to-peer) was tested as well. Latency measurement between each two nodes was performed 20 times (both initiated the connection 10 times) with the ping command and 64 bytes (with ICMP header) of data. The results of these tests are presented in Figure 19, where the X axis shows latency in milliseconds, while the Y axis shows how many times that latency occurred in all tests (the numbers of tests for Docker and Host runtime machines were the same). Docker network latency is about two times greater and less stable than for Host. The minimum, maximum and median latency values are presented in Table 10.

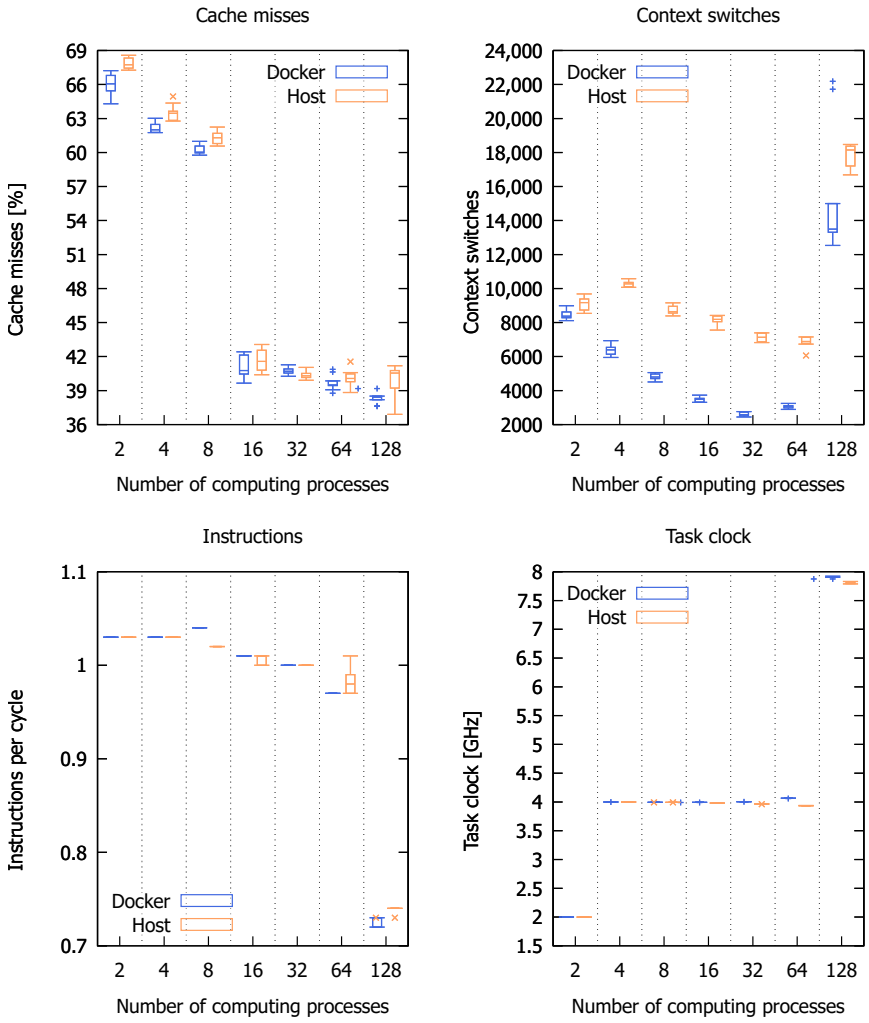

Figure 17. Hardware counters for SPMD program with -O3 and -march=native compilation parameters; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

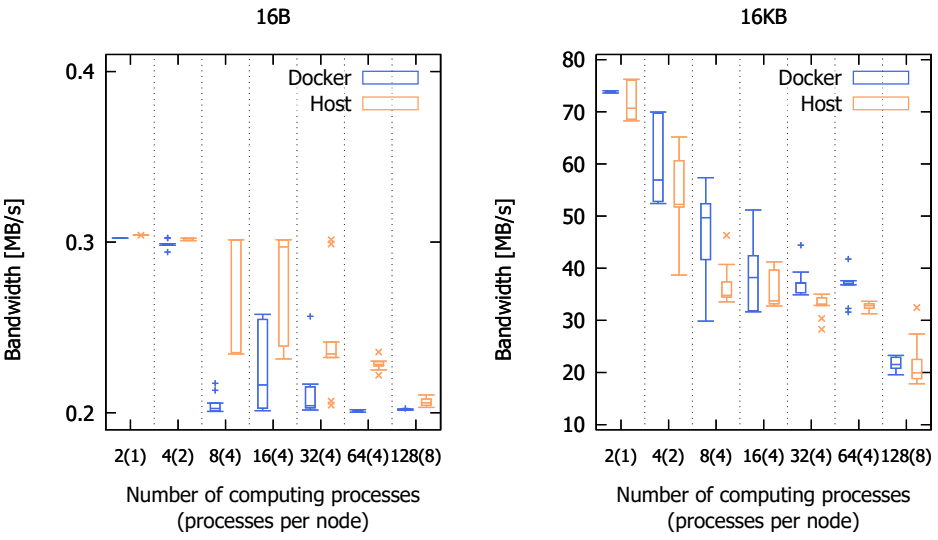

Figure 18. *Cont.*

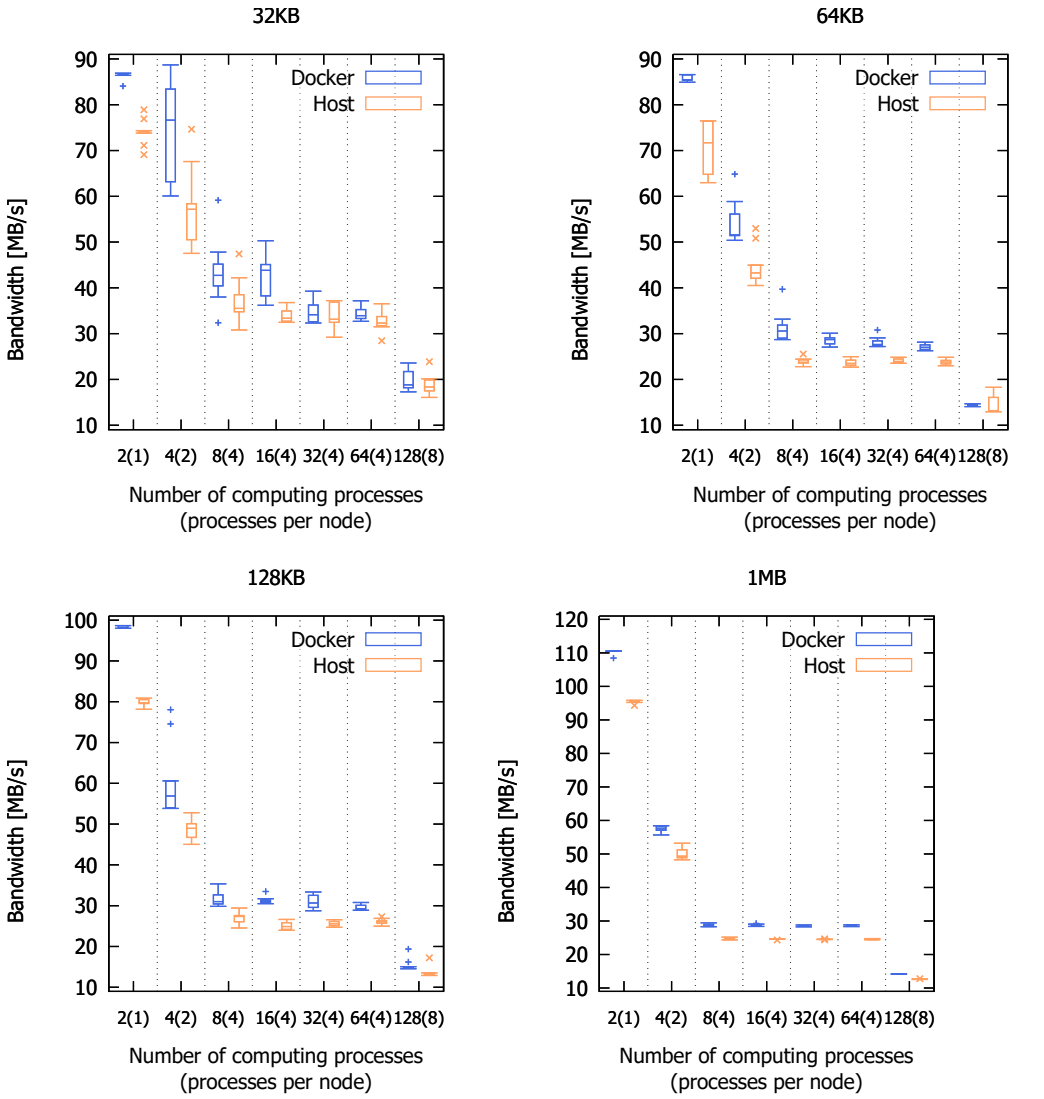

Figure 18. Network bandwidth between nodes for selected size of data; boxplot notation: medians, Q1 and Q3, max(Q1−1.5IQR, minimum) and min(Q3+1.5IQR, maximum) as well as potential outliers (+ for Docker, x for Host).

Table 10. Minimum, maximum and median communication latency between nodes for Docker and Host.

Runtime	Latency [s]		
Machine	Min	Max	Median
Docker	0.258	0.656	0.419
Host	0.136	0.361	0.232

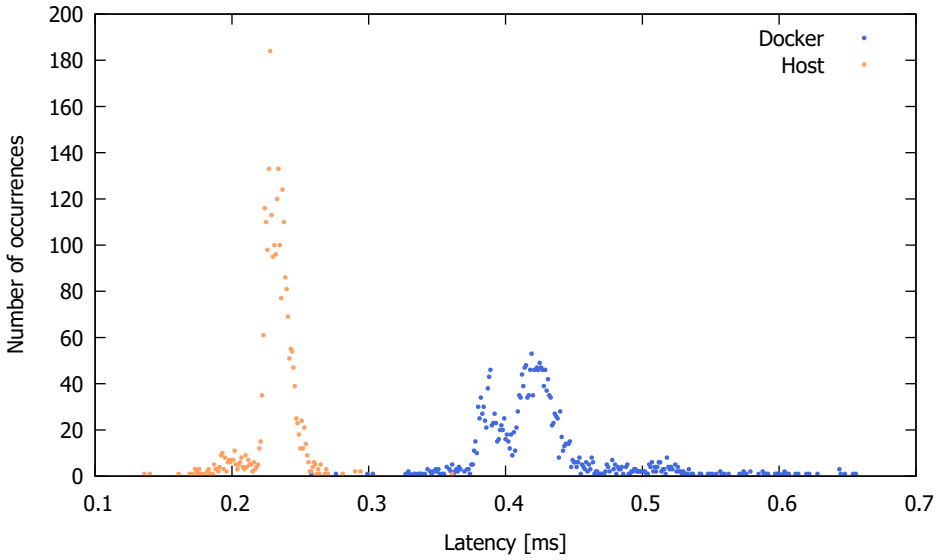

Figure 19. Communication latency between nodes for Docker and Host.

7. Summary and Future Work

As the use of containerization brings many benefits for managing computing environments, e.g., isolation, easy preparation and quick change of the runtime environment, need for assessment of its performance and comparison to a non-virtualized environment is of considerable importance. While many such comparisons have been performed in the past, comparing VMs and containers, various container solutions, in this paper, we investigated three applications representative of master–slave, geometric SPMD and divide-and-conquer codes, and focused on investigation of impact of compilation optimization flags and analysis of execution times and speed-ups for particular numbers of processes, assessment of best configurations and comparison of Docker versus bare-metal times for such preferred set-ups. We have concluded that the overhead of Docker versus bare-metal for the aforementioned applications is 7.59%, 13.29% and 15.30%, respectively, and the best compilation flags were -O3_native, -O3_native and -O2, giving shortest times on 64, 128 and 256 processes for the applications, respectively.

Furthermore, the applications tested in this work can be further tested in their optimized versions, taking into account: multithreading with MPI [29], overlapping communication and computations for master–slave and geometric SPMD versions, as well as a dynamic partitioning scheme for the divide-and-conquer approach [9]. In the latter, dynamic process spawning [29] can be used to handle imbalanced parts of the computational tree.

The Docker Swarm is one of the many container orchestration tools. The tests performed did not show that it added a significant overhead for communication and computation time. In the future, it could be worth testing other tools for container orchestration in the same way, e.g., the most popular Kubernetes.

Considering these small differences between execution time using V7 and V9 for MasterSlave and SPMD and V9 version advantage for DAC together with the common recommendations for using newer versions of the software, it may be very interesting to verify if the GCC compiler and the OpenMPI library in the latest versions will give the same results.

In our analysis, we considered standard process affinity and a multi-process MPI application. In the future, it would be interesting to extend the testbed scenario to an MPI+OpenMP parallel model in which MPI is used for internode communication while multiple threads run on CPU(s) physical/logical cores.

An additional research topic would be consideration of energy related topic in the process, specifically, power capping that has already been proven useful for optimizing

performance-energy related metrics [30]. In this context, the virtualization overhead could be considered as well.

Author Contributions: Conceptualization, P.C., T.K.; methodology, P.C.; software, P.C., T.K.; validation, P.C., T.K.; formal analysis, P.C., T.K.; investigation, T.K.; resources, P.C.; data curation, T.K.; writing—original draft preparation, P.C., T.K.; writing—review and editing, P.C., T.K.; visualization, T.K.; supervision, P.C.; project administration, P.C.; funding acquisition, P.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: https://github.com/tomaszkononowicz/HPCDockerMPIPaper, accessed on 1 April 2022.

Acknowledgments: In this work, we used facilities located at the Dept. of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Poland.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, D.; Peng, J.; Zhang, X.; You, Y.; Ning, B. Application features-based virtual machine deployment strategy in cloud environment. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e6691. [CrossRef]
2. Giallorenzo, S.; Mauro, J.; Poulsen, M.; Siroky, F. Virtualization Costs: Benchmarking Containers and Virtual Machines Against Bare-Metal. *SN Comput. Sci.* **2021**, *2*, 404. [CrossRef]
3. Kałaska, R.; Czarnul, P. Investigation of Performance and Configuration of a Selected IoT System—Middleware Deployment Benchmarking and Recommendations. *Appl. Sci.* **2022**, *12*, 5212. [CrossRef]
4. Aruna, K.; Pradeep, G. Measure The IoT Framework Using Docker With Fog Computing. *Int. J. Sci. Technol. Res.* **2020**, *9*, 5677–5682.
5. Ismail, B.I.; Mostajeran Goortani, E.; Ab Karim, M.B.; Ming Tat, W.; Setapa, S.; Luke, J.Y.; Hong Hoe, O. Evaluation of Docker as Edge computing platform. In Proceedings of the 2015 IEEE Conference on Open Systems (ICOS), Melaka, ML, USA, 24–26 August 2015; pp. 130–135. [CrossRef]
6. Saha, P.; Beltre, A.; Uminski, P.; Govindaraju, M. Evaluation of Docker Containers for Scientific Workloads in the Cloud. In Proceedings of the Practice and Experience on Advanced Research Computing, Pittsburgh, PA, USA, 22–26 July 2018.
7. Shirinbab, S.; Lundberg, L.; Casalicchio, E. Performance evaluation of containers and virtual machines when running Cassandra workload concurrently. *Concurr. Comput. Pract. Exp.* **2020**, *32*, e5693. [CrossRef]
8. Rezende Alles, G.; Carissimi, A.; Mello Schnorr, L. Assessing the Computation and Communication Overhead of Linux Containers for HPC Applications. In Proceedings of the 2018 Symposium on High Performance Computing Systems (WSCAD), Sao Paulo, Brazil, 1–3 October 2018; pp. 116–123. [CrossRef]
9. Czarnul, P. *Parallel Programming for Modern High Performance Computing Systems*; CRC Press: Boca Raton, FL, USA; Taylor & Francis: Abingdon, UK, 2018; ISBN 9781138305953.
10. Chung, M.T.; Quang-Hung, N.; Nguyen, M.T.; Thoai, N. Using Docker in high performance computing applications. In Proceedings of the 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), Ha Long, Vietnam, 27–29 July 2016; pp. 52–57. [CrossRef]
11. Potdar, A.M.; D G, N.; Kengond, S.; Mulla, M.M. Performance Evaluation of Docker Container and Virtual Machine. *Procedia Comput. Sci.* **2020**, *171*, 1419–1428. [CrossRef]
12. Benedicic, L.; Cruz, F.A.; Madonna, A.; Mariotti, K. Sarus: Highly Scalable Docker Containers for HPC Systems. In *High Performance Computing*; Weiland, M., Juckeland, G., Alam, S., Jagode, H., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 46–60.
13. Kurtzer, G.; Sochat, V.; Bauer, M. Singularity: Scientific containers for mobility of compute. *PLoS ONE* **2017**, *12*, e0177459. [CrossRef] [PubMed]
14. Gannon, D.; Sochat, V. *Cloud Computing for Science and Engineering*; Chapter Singularity: A Container System for HPC Applications; MIT Press: Cambridge, MA, USA, 2017; ISBN 978-0262037242.
15. Layton, J. Singularity—A Container for HPC. *Admin Magazine*, 2021. Available online: https://www.admin-magazine.com/HPC/Articles/Singularity-A-Container-for-HPC (accessed on 1 April 2022).
16. Di Tommaso, P.; Palumbo, E.; Chatzou, M.; Prieto, P.; Heuer, M.; Notredame, C. The impact of Docker containers on the performance of genomic pipelines. *PeerJ* **2015**, *3*, e1273. [CrossRef] [PubMed]
17. Gerber, L. *Containerization for HPC in the Cloud: Docker vs. Singularity. A Comparative Performance Benchmark*; Umea Univrsitet: Umea, Sweden, 2018.

18. Martin, J.P.; Kandasamy, A.; Chandrasekaran, K. Exploring the Support for High Performance Applications in the Container Runtime Environment. *Hum.-Centric Comput. Inf. Sci.* **2018**, *8*, 1. [CrossRef]
19. Aoyama, K.; Watanabe, H.; Ohue, M.; Akiyama, Y. Multiple HPC Environments-Aware Container Image Configuration Workflow for Large-Scale All-to-All Protein–Protein Docking Calculations. In *Supercomputing Frontiers*; Panda, D.K., Ed.; Springer International Publishing: Cham, Switzerland, 2020; pp. 23–39.
20. Noecker, C. Making Scientific Applications Portable: Software Containers and Package Managers, 2018. All College Thesis Program, 2016–2019. Volume 46. Available online: https://digitalcommons.csbsju.edu/honors_thesis/46 (accessed on 1 April 2022).
21. NVIDIA. CUDA C++ Programming Guide. 2022. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 1 April 2022).
22. Czarnul, P.; Rościszewski, P.; Matuszek, M.; Szymański, J. Simulation of parallel similarity measure computations for large data sets. In Proceedings of the 2015 IEEE 2nd International Conference on Cybernetics (CYBCONF), Gdynia, Poland, 24–26 June 2015; pp. 472–477. [CrossRef]
23. Czarnul, P.; Ciereszko, A.; Frązak, M. Towards Efficient Parallel Image Processing on Cluster Grids Using GIMP. In *Computational Science-ICCS 2004*; Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 451–458.
24. Singh, G.; Diamantopoulos, D.; Hagleitner, C.; Gómez-Luna, J.; Stuijk, S.; Mutlu, O.; Corporaal, H. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications, Gothenburg, Sweden, 31 August–4 September 2020.
25. Szustak, L.; Bratek, P. Performance portable parallel programming of heterogeneous stencils across shared-memory platforms with modern Intel processors. *Int. J. High Perform. Comput. Appl.* **2019**, *33*, 534–553. [CrossRef]
26. Czarnul, P.; Grzeda, K. Parallel Simulations of Electrophysiological Phenomena in Myocardium on Large 32 and 64-bit Linux Clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*; Kranzlmüller, D., Kacsuk, P., Dongarra, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2004; pp. 234–241.
27. Soppelsa, F.; Kaewkasi, C. *Native Docker Clustering with Swarm*; Packt Publishing: Birmingham, UK, 2017.
28. Rudyy, O.; Garcia-Gasulla, M.; Mantovani, F.; Santiago, A.; Sirvent, R.; Vázquez, M. Containers in HPC: A Scalability and Portability Study in Production Biological Simulations. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 567–577. [CrossRef]
29. Gropp, W.; Hoefler, T.; Thakur, R.; Lusk, E. *Using Advanced MPI: Modern Features of the Message-Passing Interface*; The MIT Press: Cambridge, MA, USA, 2014.
30. Krzywaniak, A.; Proficz, J.; Czarnul, P. Analyzing energy/performance trade-offs with power capping for parallel applications on modern multi and many core processors. In Proceedings of the 2018 Federated Conference on Computer Science and Information Systems, FedCSIS 2018, Poznań, Poland, 9–12 September 2018; Annals of Computer Science and Information Systems; Ganzha, M., Maciaszek, L.A., Paprzycki, M., Eds.; 2018; Volume 15, pp. 339–346. [CrossRef]