





MPI to Go: Container Clusters for MPI Applications

Luiz Angelo Steffene¹ , Andrea S. Charão² , Bruno Alves²,
Lucas R. de Araujo², and Lucas F. da Silva²

¹ CREsTIC Laboratory, Université de Reims Champagne Ardenne, Reims, France
angelo.steffene@univ-reims.fr

² Universidade Federal de Santa Maria, Santa Maria, Brazil
andrea@inf.ufsm.br, {[bdalves](mailto:bdalves@inf.ufsm.br),[lraraujo](mailto:lraraujo@inf.ufsm.br),[lferreira](mailto:lferreira@inf.ufsm.br)}@inf.ufsm.br

Abstract. Container-based virtualization has been investigated as an attractive solution to achieve isolation, flexibility and efficiency in a wide range of computational applications. In High Performance Computing, many applications rely on clusters to run multiple communicating processes using MPI (Message Passing Interface) communication protocol. Container clusters based on Docker Swarm or Kubernetes may bring benefits to HPC scenarios, but deploying MPI applications over such platforms is a challenging task. In this work, we propose a self-content Docker Swarm platform capable of supporting MPI applications, and validate it though the performance characterization of a meteorological scientific application.

Keywords: Container-based virtualization · High Performance Computing · Performance evaluation · MPI

1 Introduction

1.1 Motivation

High performance computing (HPC) is a generic term for computationally or data-intensive applications of a [25] nature. While most HPC platforms rely on dedicated and expensive infrastructures such as clusters and grids, other technologies such as cloud computing are becoming attractive. Recent developments on the virtualization domain have considerably reduced the performance overhead of these new platforms. Furthermore, traditional HPC infrastructures must often struggle with administration and development issues as the installation and maintenance of HPC applications often leads to library incompatibilities, access rights conflicts or simply dependencies problems for legacy software.

The arrival of efficient virtualization techniques such as container-based virtualization has set a new landmark towards the maintainability of computing infrastructures. Concepts such as isolation and packaging of applications now allow a user to create its own execution environment with all required libraries, to distribute this environment and to reproduce the same install everywhere with almost no effort.

When considering HPC applications, the MPI (Message Passing Interface) protocol [21] is often used for data exchange and task coordination in a cluster. Despite recent advances in its specification, the deployment of an MPI application is still too rigid to be easily deployed on more dynamic environments such as cloud or container clusters. Indeed, the starting point of an MPI execution is the definition of a list of participant nodes, which requires a prior knowledge of the runtime environment.

Deploying an MPI cluster on containers clusters is also difficult task because the internal overlay network from popular environments such as Docker Swarm is designed for load balancing and not for addressing specific nodes, as in the case of MPI. As a consequence, only a few works in the literature try to offer support MPI on Docker, and most fail to develop a simple and stand-alone solution that does not require manual or external manipulation of the MPI configuration.

In this paper we address the lack of elegant deployment solutions for MPI over a Docker Swarm cluster. We extend the preliminary results presented on [27], demonstrating the interest of our platform through the deployment of a meteorological simulation software and its evaluation thanks to execution benchmarks and trace analysis on both cloud and container environments. In addition, we expand the analysis by comparing the performance on traditional x86 processors and ARM processors represented by clusters of Raspberry Pi machines.

1.2 Background

Considering all current virtualization technologies, we can highlight two of them: hardware virtualization, which makes use of Virtual Machine Monitors (VMMs), better known as **hypervisors**, and OS-level virtualization (**containers**).

Hardware virtualization can be classified as Type I or Type II. Each type considers where the hypervisor is located in the system software stack. Type I hypervisors (Fig. 1a) execute directly over hardware and manage the guest OSs. This way, the access to the hardware (and the isolation between different host OS) must be aware of the underlying VMM to access the hardware (both through the hypervisor or through paravirtualization interfaces).

The Type II (Fig. 1b) virtualization, on the other hand, relies on a hypervisor working inside the host OS, with the later one ensuring the access to the hardware. Type II allows creating complete abstractions and total isolation from the hardware by translating all guest OS instructions [18]. This type of virtualization is also known as *full virtualization*. As drawback, it imposes a high overload that penalizes most HPC applications [33]. While these performance penalties can be mitigated by the use of hardware-assisted virtualization (a set of specific

instructions present in most modern processors), other virtualization strategies are often preferred when dealing with HPC applications.

As hypervisor-based solutions are considered heavy-weighted, the development of OS-level virtualization is becoming much more popular (Fig. 1c). This approach uses OS features that partition the physical machine resources, creating multiple isolated user-space instances (containers) on top of a single host kernel. Another advantage of container-based virtualization is that it does not need a hypervisor, incurring much less overhead [12]. Hence, popular OS-level virtualization systems include Linux Containers (LXC), RKT and Docker.

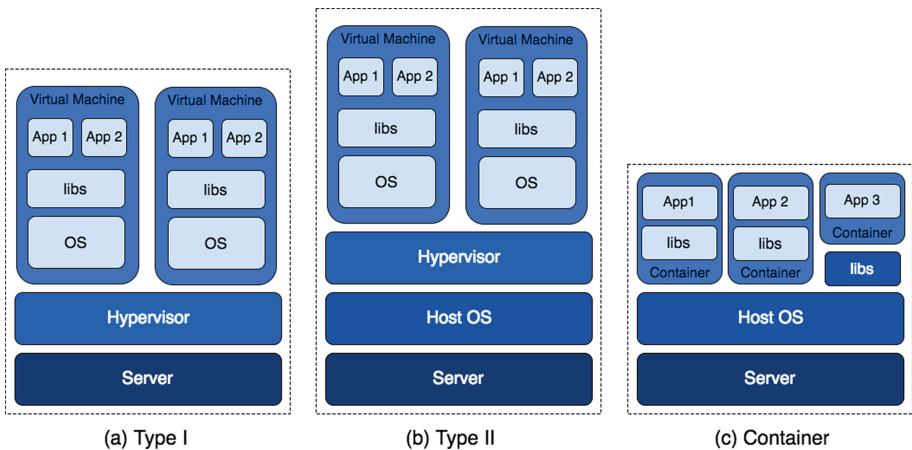


Fig. 1. Main virtualization types.

In all those systems, the container shares the kernel with the host OS, so its processes and file system are completely visible from the host, but thanks to the OS isolation, the container only sees its file system and process space [31]. They also use *namespaces* to isolate the containers and ensure that they access only their subsets of resources. Namespaces are also used to control the network and inter-process communication capabilities, and allow containers to be checkpointed, resumed or even migrated.

One of the most popular container solutions is Docker¹. In addition to managing containers at the OS level, Docker allows the users to create personalized images that can be saved and used as a base to the deployment of many concurrent containers. Docker also provides a registry-based service named *Docker Hub*², allowing users to share their images.

More recently, Docker provides a basic orchestrator service called **Docker Swarm**, that enables the deployment of a cluster of Docker nodes. While Docker

¹ <https://www.docker.com/>.

² <https://hub.docker.com/>.

Swarm is not as modular as other orchestrators like Kubernetes [7], it is simple to use, and Swarm services can be easily adapted to operate under Kubernetes.

It is also interesting to note that although Docker was initially developed for x86 platforms, it now contemplates other processor architectures. For example, the adaption to the ARM processors family started around 2014, with an initial work made by Hypriot³ for Raspberry Pi machines. More recently, Docker started to officially support ARM, and several base images on Docker Hub are now published with both x86 and ARM versions.

1.3 Related Work

HPC applications often search to solve problems that are hard to compute in a single machine due to capacity (memory, storage) or performance limitations. Distribute these applications on a cluster is often a way to increase the available resources all while trying to divide the problems in small pieces that can be processed in parallel.

Traditionally, the HPC community refuses virtualization due to the performance overheads from Type II and Type I hypervisors. However, the recent dissemination of container virtualization is changing this view. Several HPC centers favor the use of containers to simplify the resources management and to guarantee compatibility and reproducibility for the users' applications [23]. For example, the NVidia DGX servers⁴, dedicated to Deep Learning and Artificial Intelligence applications, use Docker containers to deploy the user's applications.

Several strategies can be used to distribute and coordinate HPC applications over a cluster, and one of the most popular ones is the use of message passing through the MPI (Message Passing Interface) communication protocol. Indeed, MPI provides standard operations for data exchange and task coordination, including single and multi-point communications, synchronous and asynchronous primitives, parallel I/O, etc. While recent advances have improved several performance aspects, the MPI specification is still based on "stable" HPC clusters, which may hinder the deployment of an MPI application on new dynamic environments such as clouds and virtualized clusters. Indeed, MPI requires a well-known execution environment, characterized by a list of participating nodes (often known as the `hostfile`). Please note that while some fault-tolerance has been included in the last MPI specification (see [21]), it only serves to handle nodes that go down, not to manage a dynamic list of nodes.

As a consequence, deploying MPI applications over a container cluster such as Docker Swarm or Kubernetes is a challenging task. In the specific case of Docker Swarm (which we consider in this work), the overlay network connecting the containers is designed to perform replication and load balancing, not to manage node lists as required by MPI. This problem can be found on the literature, where most works trying to deploy MPI application over Docker require manual

³ <https://blog.hypriot.com/>.

⁴ <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-centos-datasheet-update-r2.pdf>.

or external manipulation of MPI elements to deploy applications. One of our aims is therefore to propose a self-content Docker Swarm platform capable of supporting MPI applications.

Among the related works, we can cite [5], which focus on the automation of the deployment of the MPI application over a Docker Swarm cluster (i.e., scripts to copy and launch the application), but requires the user to provide the list of available nodes at launch time. More often, the literature describes an external management for the nodes in the container cluster. For example, [32] suggests two architectural approaches to be used with Docker, all relying on an external script that feeds information to the containers through SSH. The same approach is used by [13], where the containers are launched separately by PBS, a resource manager, and a set of scripts helps deploying and connecting the containers together. The same strategy is used by [3], using Slurm. In both cases, the authors connect isolated containers “by hand”, instead of relying on the Docker Swarm orchestrator.

Not all solutions rely on scripts, but sometimes they depend on solutions specifically tailored for MPI. This is the case of Singularity [16], a container manager developed for HPC applications. As these applications often rely on MPI, Singularity automatically sets up an MPI hostfile with the host names.

An extreme case of external dependency is that of [8], where even MPI tools (`mpirun`, `mpiexec`) are absent from the containers. Instead, both the application, data and libraries are imported from the host OS, with the containers playing simply the role of isolated execution environments. Such approach makes the execution totally dependent on the hosting platform, preventing the usage of any generic platform such as the cloud.

To our knowledge, only the work from [22] approaches the minimal requirements for the automatic deployment of MPI applications on a Docker Swarm cluster. This platform includes scripts for the deployment of the Docker Swarm service and creates list of nodes for the hostfile by inspecting active network connections (using `netstat`). Nonetheless, the use of `netstat` proved to be too unstable, and recent patches to the code try to correct this issue.

1.4 Our Contributions

In this paper, we consider the problem of developing a container environment capable of supporting MPI applications in both server and cluster configurations. As expressed before, we search to propose a self-content solution based on Docker Swarm, a well known tool that adds clustering capabilities to Docker containers, including distribution, load balance and overlay networking.

When regarding the solutions presented in the literature, we consider that they are too dependent on external scripts, making them inelegant and unreliable. Instead, we propose a simpler solutions that integrate well to the deployment of a Docker Swarm cluster.

The second contribution of this work is an extensive performance evaluation of a real application in both cloud and container environments. We selected the WRF (Weather Research and Forecast) model as it is both an application that

relies on MPI for distributed computing, but also because it has several install issues that favorize its distribution as a container image. The benchmarks and execution traces performed on this work allow us to better understand the impact of the container environment and the Docker Swarm overlay network, but also to identify performance bottlenecks from the WRF software that may be addressed in a future work. This extensive analysis is also an addition to the preliminary results we presented in [27].

Our final contribution is the adaption of the environment to support the execution on ARM processors, and another set of benchmarks and analysis in a cluster of Raspberry Pi. The main interest of supporting this family of processors lies on its potential and relative low cost. We performed new experiments in a Raspberry Pi 3 cluster, experimentally demonstrating that applications such as WRF can be deployed over ARM processors and produce results within acceptable time intervals and for a fraction of the cost of a traditional HPC platform.

2 Supporting MPI on a Docker Swarm Cluster

As we indicated before, most works aiming at supporting MPI on Docker rely on the users to complete the information in the `hostfile`, or require external tools to reach such objective. Only the work from [22] tries to create an automated process, but it does not works reliably enough.

Such difficulty is due to the fact that Docker Swarm was not conceived as an HPC environment, but rather as a self balancing/fault tolerant environment to deploy applications. This can be observed if we analyze the different execution modes Docker support: in the “individual” mode (i.e., the “original” mode that does not depend on Docker Swarm), a container is launched as a standalone application. In this mode, no additional interconnections to other instances is required, even if this can be made possible. In the “service” mode, which is part of the Docker Swarm configuration, instances are interconnected by a routing overlay. This overlay includes a naming service that allows services to locate each other easily (instead of using hard-coded IP addresses), but it also includes a load balance mechanism to redirect messages among instance replicas or to restart faulting instances.

As the own Docker Swarm documentation illustrates (see Fig. 2), replicas of a service instance, even if located in different nodes, can be addressed by the same name, with messages being rerouted by the “Ingress” overlay network. In this example, two *my-web* instances exist, and they can be contacted through any of the nodes from the service. This naming service simplifies the development of applications (the code only needs to indicate a *my-web* address), and the overlay performs the redirection of the messages and the eventual load balancing among the instances of a service.

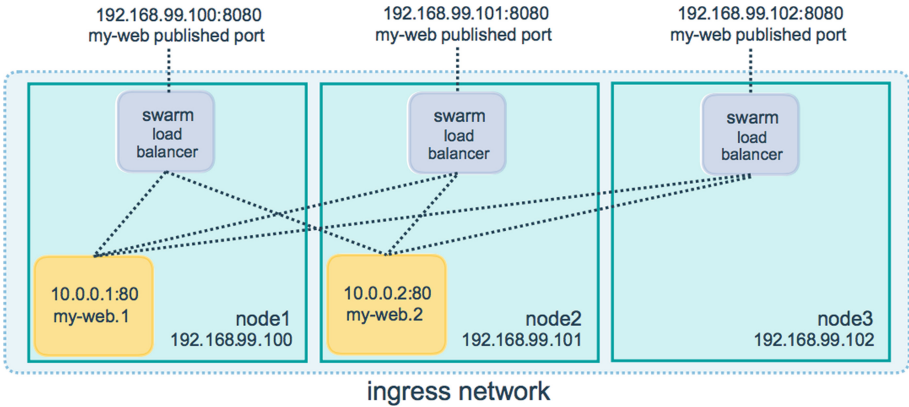


Fig. 2. Docker routing mesh [11].

In the case of MPI applications, the “replication and redirection” approach from Docker does not apply conveniently, as the MPI `hostfile` requires the list of the computing servers. The `hostfile` can be composed by hostnames or IP addresses, but obtaining the hostnames is not an easy task in Docker Swarm because the overlay hides the instances under the same “umbrella” name (for example, `my-web` in the previous example). For this reason, we need to discover the instances’ IP addresses inside the overlay network. As the approach used by [22] is not reliable as it depends on open network connections between the instances (which may be transient or even non existent), we chose to query the naming service of Docker with the `dig` DNS lookup tool. More exactly, Docker Swarm allow us to contact all instances of the same service under the `tasks.XXXX` nickname, where `XXXX` is the name of the service. By using `dig`, we can obtain detailed information sent by the DNS server and, in the case of umbrella names, get the list of the IP addresses from all instances associated with that name.

The `hostfile` also allows the users to indicate a few more information about the computing resources, like the number of processes (or slots) a node can run simultaneously. As most recent processors have multiple computing cores, we can consider that each machine in a Swarm cluster is able to run more than one process. To obtain this extra information, we call the `nproc` application on each machine, obtaining therefore the number of available processing cores.

Therefore, these two discovery steps (IP addresses and computing capabilities) can be elegantly arranged in a few lines of scripts as presented in the snippet from Fig. 3, where we compose the `hostfile` with the list of all `worker` nodes (i.e., instances of the “worker” service on Swarm) and the number of computing cores from each node.

```

iplists=`dig +short tasks.workers A`
for i in $iplists; do
    np=`ssh $i "nproc --all"`
    echo "$i:$np" >> hostfile
done

```

Fig. 3. Script to create the `hostfile` with the IP addresses and number of cores for Docker Swarm instances.

A final detail concerns the nodes' *ranks*. Most HPC applications rely on a node or process rank to perform specific tasks or to segment data to be processed, and MPI is not an exception. In MPI, the node launching the MPI application uses the order of the nodes in the `hostfile` to set the ranks and to launch the application on the other nodes. This “master” node, known as “rank 0”, is often used as a frontend node for the cluster, where the user can execute preprocessing steps, setup the application parameters or simply run the code before deploying it over the cluster. Because of this, it is important to allow users to access this node using SSH, for example. Even if most of the MPI application deployment can be automated through scripts, we believe that this improves the usability of our environment. In addition, we must consider that a Docker Swarm cluster remains an isolated environment, and accessing it via SSH is a simple way to import and export data.

Because of the load balancer in the Docker network, we cannot simply add an SSH server to each worker replica as the connection will not always be directed to the same node. Therefore, we have to create a “master” service that can be correctly identified and accessed from the outside. As the SSH port must be published, the master node cannot simply use the Ingress routing network, but needs to be executed under the special `global` deployment mode. Some other attributes in the service definition file (`docker-compose.yaml`) ensure that the master will be easy to contact (by deploying it on the manager node from the Swarm cluster), open the ports for SSH and also mount correctly all external volumes required for the application. Therefore, Fig. 4 presents the main elements of the `docker-compose.yaml` file used to define and deploy the Swarm service for our application. All these files are available at our Github repository⁵ and the images are available on Docker Hub.

⁵ https://github.com/lsteffene/swarm_mpi_basic.


```

version: "3.3"
services:
  master:
    image: lsteffenel/swarm_mpi_basic
    deploy:
      mode:
        global
      placement:
        constraints:
          - node.role == manager
    ports:
      - published: 2022
        target: 22
        mode: host
    volumes:
      - "./input:/input"
      - "./output:/output"
    networks:
      - mpinet
  workers:
    image: lsteffenel/swarm_mpi_basic
    deploy:
      replicas: 2
      placement:
        preferences:
          - spread: node.labels.datacenter
    volumes:
      - "./input:/input"
      - "./output:/output"
    networks:
      - mpinet
networks:
  mpinet:
    driver: overlay
    attachable: true

```

Fig. 4. Excerpt of the Swarm service definition.

3 Profiling and Tuning an Application

In order to assess the performance our virtualized, container-based cluster platform, we decided to benchmark Weather Research and Forecasting (WRF) model [24], a well-known numerical weather prediction model. In the next sections we will describe the WRF suite in greater details, and conduct several performance measurements and analysis to identify performance overheads and bottlenecks on both virtual cluster and application levels.

3.1 WRF - Weather Research and Forecasting

The Weather Research and Forecasting (WRF) Model [24] is a state-of-the-art numerical weather prediction software widely used for both operations, research and education. It is one of the most known meteorological forecast tools, with users numbering in the tens of thousands. Indeed, it is one of the main tools used in our MESO project⁶, an international collaboration to explore stratospheric events that affect the Ozone layer.

In spite of its popularity, WRF developers still do not offer it in binary packages ready to be installed, but instead the user needs to configure and compile the software, which may be a challenge for beginners or for users that do not have administration rights on their computing infrastructures. Today, WRF has more than 1.5 million lines of code in C and Fortran, and presents many dependencies on external software packages for input/output (I/O), parallel communications, and data compression. Many of these external libraries are becoming obsolete or unsupported by recent Linux distributions, forcing the users to download and compile these libraires too.

We believe that running WRF on containers is a way to mitigate many of the problems cited above, simplifying its deployment for both education and research usages. Indeed, containers allow the packaging of a working WRF install, ready to be used in local machines but also on the cloud.

Execution Steps. In addition to the configuration complexity, running the WRF model requires several steps to preprocess, compute and visualize the results. Indeed, the typical workflow to execute the WRF model is made of 5 phases, represented in Fig. 5 and detailed in the list below:

1. Geogrid - creates terrestrial data from static geographic data (external files with around 60 GB of data);
2. Ungrib - unpacks GRIB meteorological data obtained from an external source and packs it into an intermediate working format;
3. Metgrid - horizontally interpolates the meteorological data onto the model domain;
4. Real - vertically interpolates the data onto the model coordinates, creates boundary and initial condition files, and performs consistency checks;
5. WRF - generates the model forecast.

The three first steps belong to the WRF Preprocessing System (WPS), a subset of applications that is configured and compiled separately from the remainder of the tool. During its configuration, WPS allows two execution modes: **serial** or **dmpar**, the later one providing distributed memory parallelism through the use of MPI.

The second part of the configuration, which compiles and install the WRF model, offers four execution modes: **serial**, **smpar** (shared memory parallelism), **dmpar** (distributed memory parallelism) and **sm+dmpar**. The **smpar** option is

⁶ <http://meso.univ-reims.fr>.

based on OpenMP, while the `dmpar` uses MPI as communication overlay. The last option (`sm+dmpar`) combines OpenMP and MPI, but several user reports point out that `dmpar` usually outperforms the mixed option [9, 17] and should be preferred.

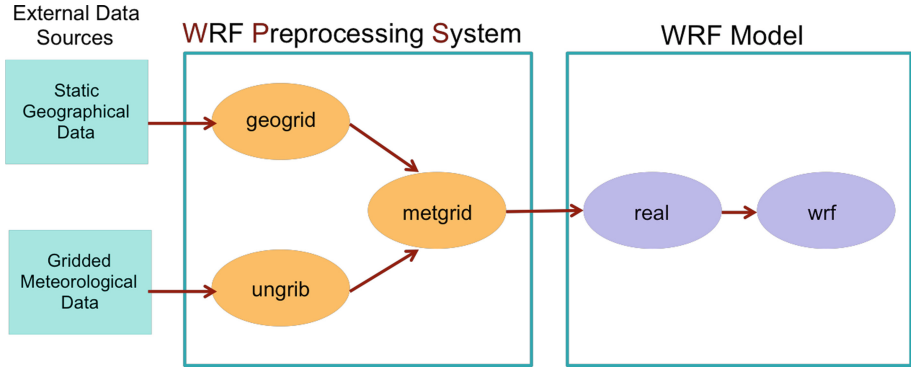


Fig. 5. WRF workflow [27].

3.2 Performance Evaluation

Computer researchers have been dedicated to investigating the impact of containers on HPC applications. The use of containers facilitates dependency management by providing a solid environment for running applications. [12, 15] have shown that the use of containers generates negligible performance payloads in a single server. In the case of Docker Swarm, however, the overlay network is created with the help of VXLAN tunnels, and this encapsulation has a payload that may represent about 6% of the transmitted data [14] and therefore a potential performance issue. Hence, in order to test the impact generated by Docker Swarm, we designed two scenarios to compare the performance of the WRF application.

To perform the tests, we used three machines configured as `c4.large` instances in the Amazon AWS service. Each machine featured a 2.9 GHz Intel Xeon E5-2666 v3 processor, 2 cores, 3.75 GiB of RAM and a network connection with moderate performance (300 Mb/s). Two scenarios were considered. In scenario (a), WRF was run on baremetal with the number of processes varying as follows: 1 (1 machine), 2 (2 machines), 4 (2 machines), 6 (3 machines). In scenario (b) the same test was performed, however, using containers with an Ubuntu OS image and with WRF installed and configured. For the execution of scenario (b), a Docker overlay network was also created so that instances could communicate, sending and receiving TCP packets used by MPI for synchronization and execution of distributed tasks. The tests on both scenarios were performed 10 times, with the average of the times presented below.

The benchmarks used a dataset concerning an area covering Uruguay and the south of Brazil and allowing a 12-h forecasting from October 18, 2016. This small dataset is used as training example for meteorology students at Universidade Federal de Santa Maria, who can modify the parameters and compare the results to the real observations. The entire dataset is accessible at the Github repository we created for the WRF container images⁷.

From the Fig. 6 chart it is possible to see that as the number of processes grows, the impact on performance from Docker is higher. This difference can be caused by the overhead generated by the overlay network, so that containers on different hosts can communicate. Nevertheless, the overall impact on performance is still reduced, reaching 7.8% when 6 processes were used. Therefore, the use of containers remains interesting, facilitating the deployment of a given application or tool. The next section conducts a detailed analysis on the execution traces of WRF, allowing us to better understand the reasons for the reduced speedup observed in both baremetal and container environments.

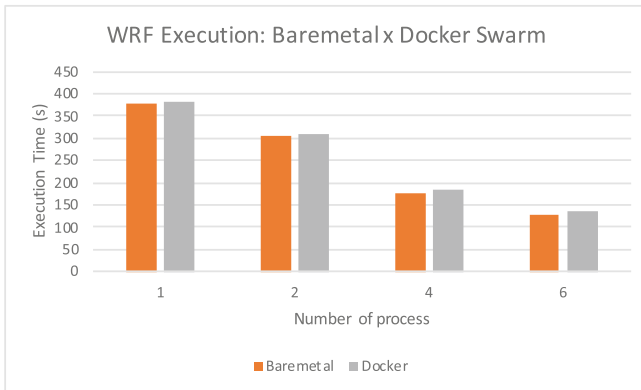


Fig. 6. WRF performance on AWS: baremetal versus Docker.

3.3 Going Deeper: Tracing MPI Communications

Parallel applications have their own circumstances when it comes to analyzing its execution behavior. Beyond the individual flow of each process, it is essential to monitor how the processes communicate with each other and how this communication affects the overall performance. The tracing tools and its outcome enables this evaluation to be done, as the traces are valuable resources to find unexpected behaviors or bottlenecks during the execution. Along with this investigation, trace visualization tools will also be helpful. These tools are crucial when it comes to realizing the *post-mortem* analysis, facilitating to identify

⁷ <https://github.com/lstefene/wrf-container-armv7l-RaspberryPi>.

events that might be affecting the performance. Lastly, it is desirable to connect these events with the code to implement improvements.

Currently, there is a variety of tracing tools available, along with a variety of trace visualization tools. Among the existent tracing tools, EZTrace appears as a tool designed for providing a generic way to analyze an application without impacting its execution [28]. Besides that, EZTrace was commonly adopted in previous works and provides runtime instrumentation. Since these characteristics fit our needs, EZTrace was chosen to trace the executions of the case study application, described in Sect. 3. The version chosen for EZTrace was version 1.1-6, that is compatible with the `binutils-dev` package, version 2.26.1-1, available on Ubuntu 16.04. This package is a requirement and its version impacts which EZTrace version to choose. To the current case study, it was necessary to include the MPI module, as we are meant to collect data in this context during the execution.

The tracing analysis was performed on the same cloud environment as the previous performance tests, using three `c4.large` on Amazon Web Services (AWS). Two cases were executed to generate traces, the first with 4 WRF processes distributed over two nodes, and the second with 6 processes distributed over three nodes, just like in Sect. 3.2. Were realized five executions for each case, and EZTrace was instrumented to collect MPI related events (`eztrace -t mpi`).

As the executions finished and the data was collected, we converted it into two trace formats: Pajé and OTF. To visualize these trace formats, ViTE⁸ (Pajé) and Vampir⁹ (OTF) were the visualization tools utilized. Each one of these tools provides specific resources to explore the traces and its data.

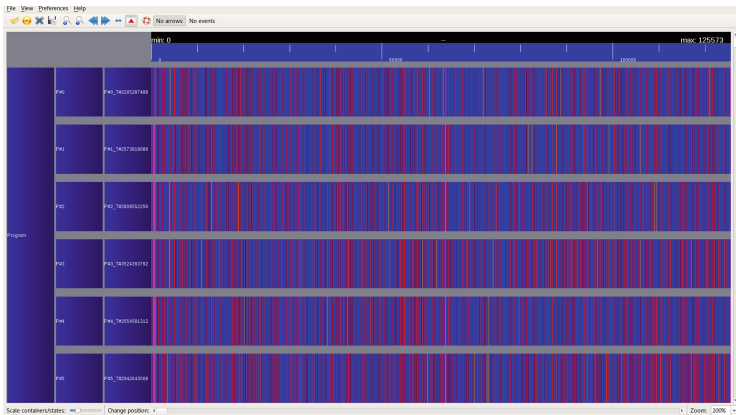


Fig. 7. ViTE tool interface.

⁸ <http://vite.gforge.inria.fr/>.

⁹ <https://vampir.eu/>.

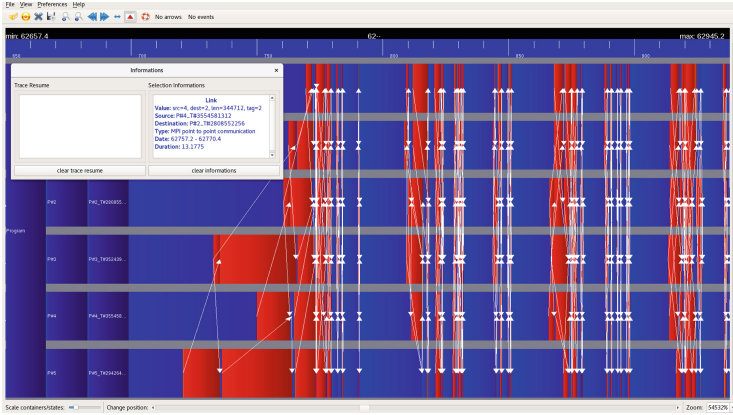


Fig. 8. ViTE zoom on communications.

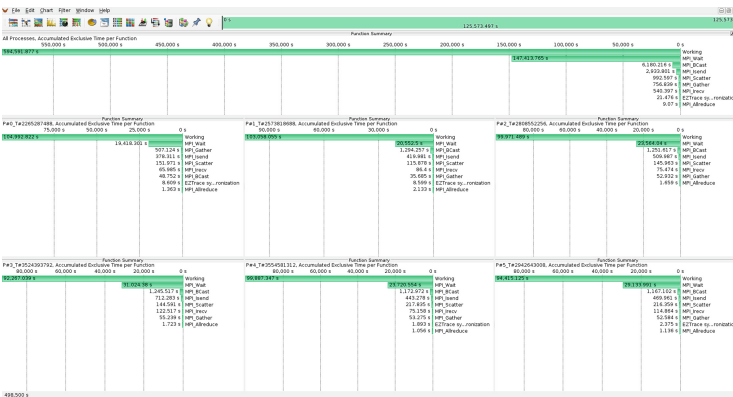


Fig. 9. Communication matrices.

Firstly, the view on ViTE tool is presented in Fig. 7, illustrating the communications between 6 WRF processes. The ViTE interface allows us to zoom in and see the messages' flow along with more details about a specific event or arrow, as the user double clicks it (Fig. 8). The tool also has a statistical plugin that presents the data collected in charts shape.

Another visualization tool utilized was Vampir. Its interface provides multiple resources to investigate the trace. These resources include most of what is available on ViTE and some extra. For instance, in Fig. 9, a function summary is shown. With this resource, it is possible to view how much utilized is each MPI function, in general, and for each process. Another resource presented, shown in Fig. 10 is a communication matrix. It enables to visualize how many times a process has sent and received messages from another process. It also shows

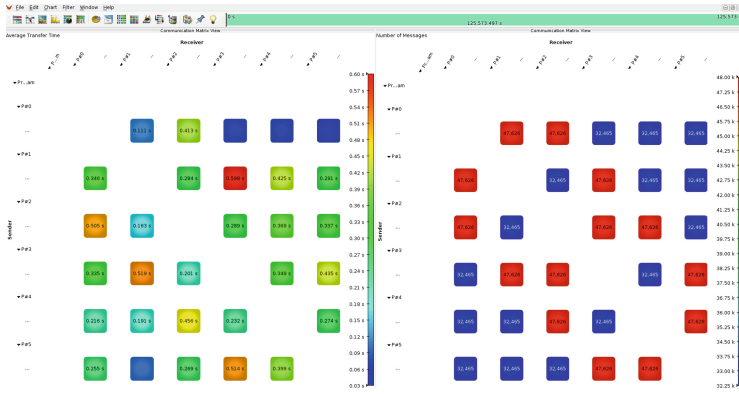


Fig. 10. Resources available on Vampir tool.

the shortest, longest, and average transfer time for each specific combination of sender and receiver process.

Looking these traces, we see that MPI_Wait events represent an important part of the execution time. MPI_Wait is a primitive used when non-blocking functions as MPI_Isend and MPI_Irecv, are called. Indeed, it indicates that a process is waiting for the arrival of a message, evidencing a lack of synchronization between processes. Even though the time spent during this invocation is inconvenient, non-blocking messages are usually an improvement over blocking messages, since non-blocking calls are used to overlap communication and computation. This behavior allows processing between the time that the process sends the message and the MPI_Wait instance but, in the case of the WRF application, it seems too important and therefore affect the overall performance.

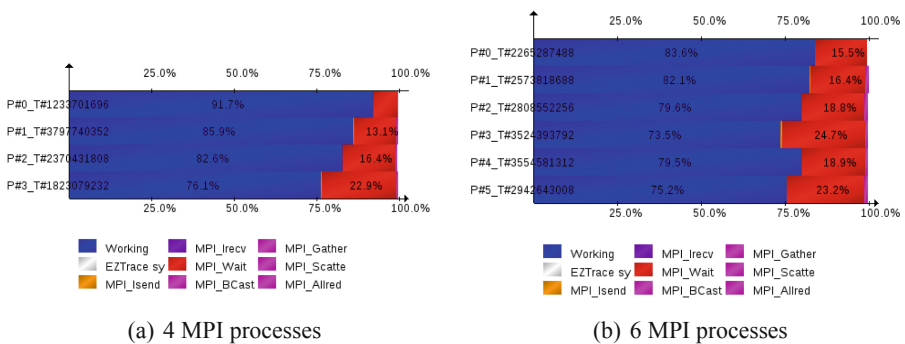


Fig. 11. Performance statistics.

Hence, the time occupied by MPI_Wait increases with the number of processes, as we observe in the statistical details obtained with 4 and 6 processes

on Fig. 11. Also, the time expended in `MPI_Wait` is not equally distributed, evidencing a load balance issue inside WRF.

While this analysis does not uncover any special problem with the container environment, it helps concluding that scalability issues observed in Sect. 3.2 are due to the own WRF code, whose solution is beyond the scope of this work.

4 Supporting Different Processor Architectures

4.1 WRF on ARM Architectures

In the previous sections we described the design of a Docker Swarm platform supporting MPI applications, and we perform benchmarks using the WRF meteorological model as a testing subject. Those benchmarks were performed in classical x86 processors, which are popular among the HPC community.

In recent years, however, we have seen the arrival of computing platforms based on ARM processors. Most ARM computers are based in the System-on-a-Chip (SoC) model that encapsulates CPU, GPU, RAM memory and other components on the same chip [30], including the popular single-board computers like Raspberry Pi, but also cellular phones and tablets.

ARM processors are currently used for a large range of applications, from Computer Science teaching [1] to Internet of Things [19]. They have an active role in Fog and Edge computing [26], bringing computation closer to the user and therefore offering proximity services that otherwise would be entirely deployed on a distant infrastructure. All naturally, many cloud providers such as Amazon and Google start to propose servers running on ARM processors.

The HPC community has demonstrated an increasing interest in this architecture, with many projects on the way. If the choice for ARM processors was initially driven by energy and cost requirement, nowadays this family of processors presents several improvements that allow the construction of computing infrastructures with a good computing power and a cost way inferior to traditional HPC platforms [10, 20, 29]. Furthermore, a SoC cluster can substitute a traditional HPC cluster in some situations, as SoC are relatively inexpensive and have low maintenance and environmental requirements (cooling, etc.). Of course, this is only valid as long as the SoC infrastructure provides sufficient Quality of Service (QoS) to the final users.

The use of Docker on SoCs represent also an interesting solution to deploy scientific applications for educational purposes [2]. Indeed, if virtualization (and especially **container-based virtualization**) contributes to simplify the administrative tasks related to the installation and maintenance of scientific applications, it also enables a rich experimental learning for students, which can test different software and perform hands-on exercises without having to struggle with compilers, operating systems, and DevOps tasks. Furthermore, by developing solutions for both x86 and ARM architectures, we try to simplify the deployment of applications on personal computers, classrooms, dedicated infrastructures or even the cloud, seamlessly. While the MPI support is almost identical in both architectures, we faced some additional difficulties when adapting

WRF. When we started to configure WRF for the ARM platform, we had to address a few additional issues besides the availability of some libraries, a problem already observed during the configuration on x86. Indeed, the configuration of WRF supports several compilers (gcc, Intel, Portland, etc.) and architectures, but ARM processors are not listed among the supported ones. Fortunately, some researchers had the same problem before and documented their experiences, like for example the work from [4]. While the adaption requires the editing of the configuration files in order to find a match to the ARM platform, the configuration differences for both ARM or x86 are minimal, and most of the process is simple and straightforward.

In addition, we have changed how to access input data, from a fixed Docker volume to a mounted file system. This gives more flexibility to develop workflows to execute the application regularly, like for example in a daily forecast schedule. This also helps to fix the problems due to the storage of the *geogrid* geographical database. As the full database reaches 60 GB when uncompressed, the users can attach an external storage drives to their nodes instead of having all the database in the Docker image.

In order to assess the interest running meteorological simulations on ARM processors, we conducted a series of experiments to evaluate the performance of each step of the WRF application. The next sessions describe the experiments and present our first insights.

4.2 Environment Description

In these benchmarks we deployed our virtual cluster over a network of three **Raspberry Pi 3 model B** (Broadcom BCM2837 processor, ARM Cortex-A53, 4 cores, 1.2 GHz, 1 GB RAM). The interconnection between devices is by a Fast Ethernet switch (100 Mbps). As the goal of the cluster is to run an application that demands considerable computing resource, for reduce the effects of limited RAM memory, to provide a Swap memory, a USB drive (1.8 GB) has been connected to each device. The WRF dataset is the same used on the experiments from Sect. 3.2.

All measures presented in this section correspond to the average of at least 5 runs. Furthermore, as the WRF workflow is composed by 5 steps, we computed the execution time of each step individually, in order to determine the best deployment strategy. Therefore, the next sections present the separate analysis of the preprocessing steps (all three steps from WPS and the real step from WRF) and the forecast step (WRF).

4.3 Performance of WPS and Real Steps

In traditional x86 environments, the execution of the WRF workflow is dominated by the WRF model: all WPS preprocessing steps (*geogrid*, *ungrib*, *metgrid*) and the *real* steps represent only a small fraction of the computing time. On ARM processors (and especially on SoCs) this is not necessarily

true, and we need to identify the constraints we can face during the usage of those machines.

One of the first issues concern the access to the geographical database required by the `geogrid` step. As indicated in Sect. 3.1, this database has more than 60 GB, which is a potential problem for the internal storage of a typical SoC that relies on SD cards. Indeed, in our experiments, we had to attach an external USB storage device to a Raspberry Pi node to accommodate this WPS_GEOG database.

As the ARM processors used in our experiments are less powerful than x86 processors, we decided to study the performance of each workflow step separately, trying to identify whether the use of MPI would benefit each one of the WPS steps (as well as the real step). For such, we measured the execution time of each step when varying the number of computing cores (using the `mpirun -np` option).

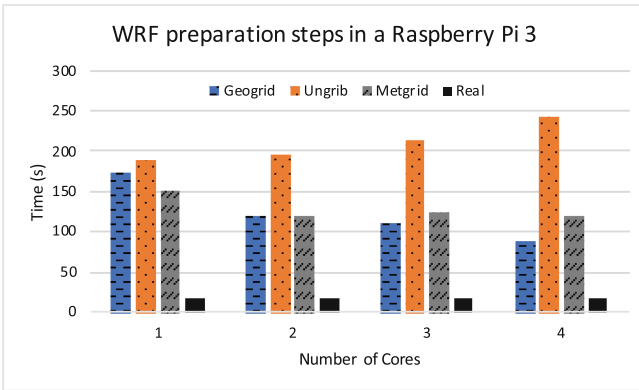


Fig. 12. Performance of WPS steps when varying the number of cores [27].

Table 1. Relative performance of WPS steps on a single machine (in seconds) [27].

Cores	1	2	3	4
Geogrid	173.81	119.59	111.56	88.54
Ungrib	188.78	196.15	212.97	241.57
Metgrid	151.42	120.47	123.56	119.26
Real	16.437	16.54	16.59	16.69

The result of this benchmark, illustrated in Fig. 12 and detailed in Table 1, indicates that only the *Geogrid* step effectively benefits from a multi-core execution. Even though, the acceleration is under-optimal (we need 4x cores to

obtain only a 50% performance improvement). Associated with the storage limitations cited before and its relatively small impact to the overall execution time (when comparing with the forecast step, see Sect. 4.4), we advise against running *Geogrid* cluster-wide. Instead, we suggest assigning a single node (the **master**) who can preprocess the data for the forecast model.

For all the other steps, a parallel execution is not an interesting option. The *Metgrid* step shows a small performance gain when parallelizing but the execution time stabilizes for 2 or more cores, and the *Real* step shows no evidence of improvements. In the case of *Ungrib*, the parallel execution even penalizes the algorithm. Additional benchmarks on the network performance, such as those conducted by [32], may also help tuning the different steps.

From these results, we suggest organizing the deployment of the preprocessing steps as follows:

- *Geogrid* - parallel execution with **mpirun**, preferentially only in the machine hosting the WPS_GEOG database (the **master** node);
- *Ungrib* - serial execution in a single core;
- *Metgrid* - serial execution or at most parallel execution with **mpirun** in a single machine;
- *Real* - serial execution in a single core.

4.4 WRF Execution

Even in an ARM processor, the preprocessing steps listed before represent only a small part of the execution. This is not the case of the WRF forecast step, which can have a much larger duration, especially in “production” environments with larger datasets and more than a simple 12-h forecast to be computed.

As expected, the forecasting step of WRF does benefit from multicore and cluster scenarios. Figure 13 shows the average execution time of this step when running on one, two or three nodes in the Raspberry Pi cluster.

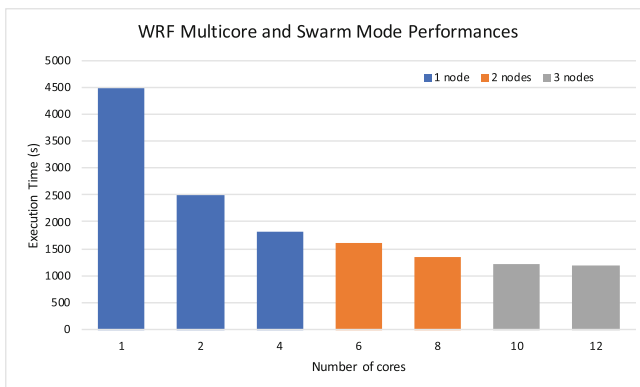


Fig. 13. Performance of WRF in multicore and Swarm cluster mode.

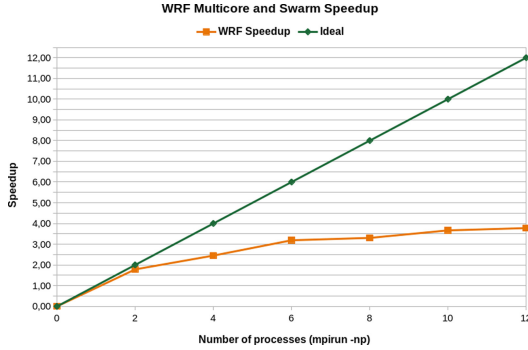


Fig. 14. Speedup of WRF execution in multicore and Swarm cluster mode.

If the multicore execution allows an important performance gain, the Swarm cluster execution shows more mitigated results. Indeed, the performance gain when passing from one to two nodes (4 to 8 process) is only 25%, and when passing from two to three nodes (8 to 12 process) it is barely 12%. As shown in Fig. 14, this is really far from a linear speedup, but can be explained both by the poor performance of WRF on `dmpar` mode (MPI) identified in Sect. 3.3. In addition, we suspect that the network performance on the Raspberry Pi also play a role. Indeed, as observed by [6], the access to the communication bus in the Raspberry Pi is known by its “low” speed interconnection card (10/100 Mbps only) that penalizes all communication interfaces.

Tables 2 and 3 detail these results, and also present a performance comparison with a the x86 processors from the AWS cloud used in Sect. 3.2. Please note that the x86 column is limited to a single `c4.large` instance from AWS as it has only two available vCPUs. Without surprise, the x86 processors are faster, but the execution time on the Raspberry Pi is still acceptable. Indeed, the processing time is fair enough for education and training. Even a production environment can be considered, if we expect WRF to deliver daily or even hourly forecasts. Furthermore, if we consider the material and environmental cost of the SoC solution, it is clearly an interesting alternative.

Table 2. WRF relative performance on a single machine (in seconds).

Cores	R Pi 3	AWS c4.large
1	4469.8374	539.05
2	2503.3624	314.69
3	2194.1872	–
4	1823.8314	–

Table 3. Performance on a Raspberry Pi 3 swarm cluster (in seconds).

Machines	Cores	Pi Swarm
$1 \times \text{Pi 3}$	1	4469.8374
	2	2503.3624
	4	1823.8314
$2 \times \text{Pi 3}$	6	1401.0106
	8	1352.72
$3 \times \text{Pi 3}$	10	1218.7158
	12	1183.2734

5 Conclusions and Future Work

This work focuses on the design of a container-based platform for HPC applications based on MPI. Indeed, container virtualization enables the packaging of complex applications and their seamless deployment. As most traditional scientific applications rely on MPI for scalability, we were surprised by the lack of a proper support for MPI, neither on popular container managers like Docker, nor on other works from the literature. We therefore propose, in a first moment, a service specification to deploy a Docker Swarm cluster that is ready for MPI applications.

Later, we evaluate the proposed platform through the performance analysis of the WRF meteorological forecast model. Through performance benchmarks on both baremetal and container environments, we were able to separate performance overheads related to the use of the container environment from those related to the own application. For instance, the analysis of the execution traces from WRF allowed us to identify performance bottlenecks that affect the scalability of the application.

Finally, we conducted a few more experiences to evaluate the performance and the interest of using containers over SoC (System-on-Chip) clusters. These results indicate that if popular the ARM processors in SoCs such as Raspberry Pi cannot compete in performance with x86 processors, they still are able to deliver results within an acceptable delay.

Future improvements to this work include the execution of additional benchmarks to validate the scalability of the platform with other MPI-based applications. Also, the bottlenecks we identified during the analysis of the execution traces of WRF will be the subject of a performance tuning project that we shall conduct in the next months.

Acknowledgements. This research has been partially supported by the French-Brazilian CAPES-COFECUB MESO project (<http://meso.univ-reims.fr>) and the GREEN-CLOUD project (<http://www.inf.ufrgs.br/greencloud/>) (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

References

1. Ali, M., Vlaskamp, J.H.A., Eddin, N.N., Falconer, B., Oram, C.: Technical development and socioeconomic implications of the Raspberry Pi as a learning tool in developing countries. In: Computer Science and Electronic Engineering Conference (CEEC), pp. 103–108. IEEE (2013)
2. Alvarez, L., Ayguade, E., Mantovani, F.: Teaching HPC systems and parallel programming with small-scale clusters. In: 2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC), pp. 1–10, November 2018. <https://doi.org/10.1109/EduHPC.2018.00004>
3. Azab, A.: Enabling Docker containers for high-performance and many-task computing. In: 2017 IEEE International Conference on Cloud Engineering (IC2E), pp. 279–285, April 2017. <https://doi.org/10.1109/IC2E.2017.52>
4. Barker, D.: Setting up an ARM-based micro-cluster and running the WRF weather model, March 2014. <http://supersmith.com/site/ARM.html>
5. de Bayser, M., Cerqueira, R.: Integrating MPI with Docker for HPC. In: 2017 IEEE International Conference on Cloud Engineering (IC2E), pp. 259–265, April 2017. <https://doi.org/10.1109/IC2E.2017.40>
6. Beserra, D., Pinheiro, M.K., Souveyet, C., Steffemel, L.A., Moreno, E.D.: Performance evaluation of OS-level virtualization solutions for HPC purposes on SOC-based systems. In: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), pp. 363–370, March 2017. <https://doi.org/10.1109/AINA.2017.73>
7. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. *Commun. ACM* **59**(5), 50–57 (2016). <https://doi.org/10.1145/2890784>
8. Chung, M.T., Quang-Hung, N., Nguyen, M., Thoai, N.: Using docker in high performance computing applications. In: 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE), pp. 52–57, July 2016. <https://doi.org/10.1109/CCE.2016.7562612>
9. HPC Advisory Council: Weather research and forecasting (WRF): performance benchmark and profiling, best practices of the HPC advisory council. Technical report, HPC Advisory Council (2010). http://www.hpcadvisorycouncil.com/pdf/WRF_Analysis_and_Profiling_Intel.pdf
10. Cox, S.J., Cox, J.T., Boardman, R.P., Johnston, S.J., Scott, M., O'brien, N.S.: Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Comput.* **17**(2), 349–358 (2014). <https://doi.org/10.1007/s10586-013-0282-7>
11. Docker Inc.: Use swarm mode routing mesh. <https://docs.docker.com/engine/swarm/ingress/>
12. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. IBM technical report RC25482 (AUS1407-001), Computer Science (2014)
13. Higgins, J., Holmes, V., Venters, C.: Orchestrating docker containers in the HPC environment. In: Kunkel, J.M., Ludwig, T. (eds.) *ISC High Performance 2015*. LNCS, vol. 9137, pp. 506–513. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20119-1_36
14. Iveson, S.: TCP/IP over VXLAN bandwidth overheads, March 2014. <https://packetpushers.net/vxlan-udp-ip-ethernet-bandwidth-overheads/>
15. Joy, A.M.: Performance comparison between Linux containers and virtual machines. In: *International Conference on Advances in Computer Engineering and Applications (ICACEA)*, pp. 342–346. IEEE (2015)

16. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: scientific containers for mobility of compute. *PLoS ONE* **12**(5), 1–20 (2017). <https://doi.org/10.1371/journal.pone.0177459>
17. Langkamp, T., Böhner, J.: Influence of the compiler on multi-CPU performance of WRFv3. *Geosci. Model Dev.* **4**(3), 611–623 (2011). <https://doi.org/10.5194/gmd-4-611-2011>. <https://www.geosci-model-dev.net/4/611/2011/>
18. Manohar, N.: A survey of virtualization techniques in cloud computing. In: Chakravarthi, V., Shirur, Y., Prasad, R. (eds.) *VCASAN-2013. LNEE*, vol. 258, pp. 461–470. Springer, Cham (2013). https://doi.org/10.1007/978-81-322-1524-0_54
19. Molano, J.I.R., Betancourt, D., Gómez, G.: Internet of Things: a prototype architecture using a Raspberry Pi. In: Uden, L., Heričko, M., Ting, I.-H. (eds.) *KMO 2015. LNBIP*, vol. 224, pp. 618–631. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21009-4_46
20. Montella, R., Giunta, G., Laccetti, G.: Virtualizing high-end GPGPUS on ARM clusters for the next generation of high performance cloud computing. *Cluster Comput.* **17**(1), 139–152 (2014). <https://doi.org/10.1007/s10586-013-0341-0>
21. MPI Forum: MPI: A message-passing interface standard version 3.1. <https://www.mpi-forum.org/docs/>
22. Nguyen, N., Bein, D.: Distributed MPI cluster with Docker Swarm mode. In: 2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC), pp. 1–7, January 2017. <https://doi.org/10.1109/CCWC.2017.7868429>
23. Ruiz, C., Jeanvoine, E., Nussbaum, L.: Performance evaluation of containers for HPC. In: HunoldHunold, S., et al. (eds.) *Euro-Par 2015. LNCS*, vol. 9523, pp. 813–824. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_65
24. Skamarock, W.C., et al.: A description of the advanced research WRF version 3. NCAR technical note. National Center for Atmospheric Research, Boulder, Colorado, USA (2008)
25. Somasundaram, T.S., Govindarajan, K.: CLOUDRB: a framework for scheduling and managing High-Performance Computing (HPC) applications in science cloud. *Future Gen. Comput. Syst.* **34**, 47–65 (2014)
26. Steffanel, L.A., Kirsch Pinheiro, M.: When the cloud goes pervasive: approaches for IoT PaaS on a ubiquitous world. In: Mandler, B., et al. (eds.) *IoT360 2015. LNICST*, vol. 169, pp. 347–356. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47063-4_36
27. Steffanel, L.A., Charão, A.S., da Silva Alves, B.: A containerized tool to deploy scientific applications over SOC-based systems: the case of meteorological forecasting with WRF. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pp. 561–568. INSTICC, SciTePress (2019). <https://doi.org/10.5220/0007799705610568>
28. Trahay, F., Rue, F., Faverge, M., Ishikawa, Y., Namyst, R., Dongarra, J.: EZTrace: a generic framework for performance analysis. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 618–619. IEEE (2011)
29. Weloli, J.W., Bilavarn, S., Vries, M.D., Derradji, S., Belleudy, C.: Efficiency modeling and exploration of 64-bit ARM compute nodes for exascale. *Microprocess. Microsyst.* **53**, 68–80 (2017). <https://doi.org/10.1016/j.micpro.2017.06.019>. <http://www.sciencedirect.com/science/article/pii/S0141933116304537>
30. Wolf, W., Jerraya, A.A., Martin, G.: Multiprocessor system-on-chip (MPSoC) technology. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **27**(10), 1701–1713 (2008)

31. Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A.F.: Performance evaluation of container-based virtualization for high performance computing environments. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 233–240, February 2013. <https://doi.org/10.1109/PDP.2013.41>
32. Yong, C., Lee, G.W., Huh, E.N.: Proposal of container-based HPC structures and performance analysis. *J. Inf. Process. Syst.* **14**(6), 1398–1404 (2018)
33. Younge, A.J., Henschel, R., Brown, J.T., von Laszewski, G., Qiu, J., Fox, G.C.: Analysis of virtualization technologies for high performance computing environments. In: IEEE 4th International Conference on Cloud Computing, CLOUD 2011, pp. 9–16. IEEE Computer Society, Washington, DC (2011)