A Containerized Tool to Deploy Scientific Applications over SoC-based Systems: The Case of Meteorological Forecasting with WRF

Luiz Angelo Steffenel[1], Andrea Schwertner Charão[2] and Bruno da Silva Alves[2]
[1]*CReSTIC Laboratory, Université de Reims Champagne-Ardenne, Reims, France*
[2]*LSC Laboratory, Universidade Federal de Santa Maria, Santa Maria, Brazil*

Keywords: Application Containers, High Performance Computing, Systems-on-a-Chip.

Abstract: Container-based virtualization represents a flexible and scalable solution for HPC environments, allowing a simple and efficient management of scientific applications. Recently, Systems-on-a-Chip (SoC) have emerged as an alternative to traditional HPC clusters, with a good computing power and low costs. In this paper, we present how we developed a container-based solution for SoC clusters, and study the performance of WRF (Weather Research and Forecasting) in such environments. The results demonstrate that although the peak performance of SoC clusters is still limited, these environments are more than suitable to scientific application that have relaxed QoS constraints.

1 INTRODUCTION

High Performance Computing (HPC) is a generic term to applications that are computationally intensive or data intensive in nature (Somasundaram and Govindarajan, 2014). While most HPC platforms rely on dedicated and expensive infrastructures such as clusters and computational grids, other technologies like cloud computing and systems-on-a-chip (SoC) are becoming interesting alternatives for HPC.

Indeed, cloud computing has brought a non-negligible flexibility and scalability for most users (Marathe et al., 2014), and a smaller maintenance cost. One drawback, however, is that the widespread of cloud computing forced a paradigm shift as applications are no longer executed directly on bare-metal but instead must be executed on top of a virtualization layer. While the performance overhead of virtualization is being rapidly reduced, it is still perceptible and may compromise some applications (Younge et al., 2011). Another inconvenience of cloud computing is that not all applications are prone to a distant execution. Latency-sensitive applications or applications executed in remote locations with limited Internet access may be penalized by a remote execution, as well as applications relying on sensitive data that cannot be transmitted to a third-part facility.

Systems-on-a-chip (SoC), on the other hand, represent a rupture on the traditional HPC infrastructure as SoC encapsulate CPU, GPU, RAM memory and other components at the same chip (Wolf et al., 2008). Most of the times, the SoC technology is used as a way to reduce the cost of single-board computers, like Raspberry Pi, ODroid or Banana Pi. These systems are currently used for a large range of applications, from Computer Science teaching (Ali et al., 2013) to Internet of Things (Molano et al., 2015). Being mostly based on ARM processors, SoCs also benefit from the improvements to this family of processors. Indeed, 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 (Weloli et al., 2017; Cox et al., 2014; Montella et al., 2014).

SoC also have an active role in Fog and Edge computing (Steffenel and Kirsch-Pinheiro, 2015), bringing computation closer to the user and therefore offering proximity services that otherwise would be entirely deployed on a distant infrastructure. 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.

In this context, the association of SoC and virtualization represents also an interesting solution to deploy scientific applications for educational pur-

Steffenel, L., Charão, A. and Alves, B.
A Containerized Tool to Deploy Scientific Applications over SoC-based Systems: The Case of Meteorological Forecasting with WRF.
DOI: 10.5220/0007799705610568
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 561-568
ISBN: 978-989-758-365-0

poses (Alvarez et al., 2018). 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 focusing on SoC, we try to minimize the material requirements to execute an application, enabling the deployment of applications on personal computers, classrooms, dedicated infrastructures or even the cloud, seamlessly.

This work is structured as follows: Section 2 reviews some elements of virtualization, while Section 3 presents how HPC applications can be challenging in a Docker environment, specially when they are based on the Message Passing Interface (MPI) standard. Section 4 introduces the WRF forecasting model and the adaptions we made to develop WRF Docker images for SoCs. Section 5 presents some benchmarking results obtained and finally Section 6 presents the conclusions obtained from this study and our plans for future works.

2 BACKGROUND

The development of OS-level virtualization is increasingly popular. This virtualization approach relies on OS facilities that partition the physical machine resources, creating multiple isolated user-space instances (containers) on top of a single host kernel. Another advantage of such **container-based virtualization** approach is that there is no execution overhead, as OS-level virtualization does not need a hypervisor (Felter et al., 2014).

One of the most popular container solutions is Docker[1]. Docker allows the creation of personalized images that can be used as a base to the deployment of many concurrent containers. While the initial releases of Docker made use of LXC as execution driver, it eventually implemented its own execution driver (Morabito et al., 2015). Docker also provides a registry-based service named *Docker Hub*[2] that allows users to share their images, simplifying the deployment of virtual images. Also, Docker provides a basic orchestrator service called **Docker Swarm** that enables the deployment of a cluster of docker nodes. While Docker Swarm is not as rich as other orchestrators like Kubernetes (Burns et al., 2016), it is simple to use, and Swarm services can be easily adapted to operate under Kubernetes.

If traditionally the HPC community was reluctant to virtualization because of the performance penalties it could incur, the dissemination of container virtualization is changing this view. More and more HPC centers favor the use of containers to simplify the resources management and to guarantee compatibility and reproducibility for the users' applications (Ruiz et al., 2015). For example, the NVidia DGX servers[3], dedicated to Deep Learning and Artificial Intelligence applications, use Docker containers to deploy the user's applications.

Although Docker was initially developed for x86 platforms, its adaption to other processor architectures like ARM started around 2014, with an initial adaption made by Hypriot[4] 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.

3 HPC ON DOCKER: LEVERAGING MPI

When considering large-scale HPC applications, they mostly rely on MPI for data exchange and task coordination across a cluster, grid or cloud. In spite of recent advances in its specification, the deployment of an MPI application can be quite rigid as it requires a well-known execution environment. Indeed, the starting point for an MPI cluster is the definition of a list of participating nodes (often known as the `hostfile`), which imposes a previous knowledge of the computing environment.

Deploying an MPI cluster over containers is a challenging task as the overlay network on Docker is designed to perform load balancing, not to address specific nodes as in the case of MPI. Most works that propose Docker images for MPI fail to develop a self-content solution, requiring manual or external manipulation of MPI elements to deploy applications. Indeed, (d. Bayser and Cerqueira, 2017) automates the deployment of the MPI application over a Docker Swarm cluster but requires the user to provide the list of available nodes (and cores) as a command-line parameter.

The external management of containers is fre-

[1]https://www.docker.com/

[2]https://hub.docker.com/

[3]https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-1/dgx-1-rhel-centos-datasheet-update-r2.pdf

[4]https://blog.hypriot.com/

quently cited as a possible solution for accommodating MPI over containers. (Yong et al., 2018) briefly describes two architectural arrangements that could be used with Docker, all relying on external automation with scripts and SSH connections to the containers. This is indeed the case of (Higgins et al., 2015), where the container orchestration is replaced by a combination of a resource manager (PBS) and a set of scripts that deploy individual container images then set them together. A similar approach is used by (Azab, 2017), who uses Slurm as orchestrator.

Another example of "external management" is the case of Singularity (Kurtzer et al., 2017), a container manager specifically designed for the HPC community although compatible with Docker images. Singularity developed a specific solution for MPI deployments, where an external tool deploys and set up the MPI hostfile, as well as copying the required application and data to the containers. In (Chung et al., 2016), MPI is not even part of the container but is mounted from the host OS, making their solution totally dependent on the execution platform.

Finally, (Nguyen and Bein, 2017), propose a generic service for the deployment of MPI applications on Docker in one machine or in a cluster, with Docker Swarm. Based on the Alpine Linux distribution, this platform automatizes most of the deployment of the Docker Swarm service, and the list of working nodes for the hostfile is obtained through the surveillance of active connections (using `netstat`). The choice of using `netstat` proved to be too unstable, and we were unable to make it work properly on a SoC.

As the existing solutions either require too much manual intervention or are not reliable enough, we decided to develop our own solutions to deploy an MPI on a Docker Swarm cluster made of SoC. Therefore, using the work from (Nguyen and Bein, 2017) as a starting point, we tried to automate the deployment of MPI as follows.

3.1 Hostfile

As explained before, most works proposing MPI over Docker delegate the task of defining the hostfile to the users. The only exception is the work from (Nguyen and Bein, 2017), who present an automated process that unfortunately does not work reliably enough.

The main reason for such difficulty is the fact that Docker presents two different execution modes that are quite dissimilar: in the "individual" mode, a container instance is launched as a standalone application, requiring no additional interconnections to other instances (although this is possible). In the "service" mode, different instances are bound together by a routing mesh and a naming service whose main purpose is to load balance messages to a given service and to easily redirect messages in case of failures. As illustrated by Figure 1, different instances can be addresses by the same name (*my-web*, in the example), simplifying the development of application that do not need any more to keep a trace of the servers' IP addresses.

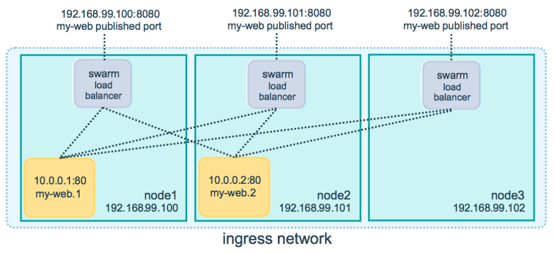

Figure 1: Docker routing mesh.

Unfortunately, the MPI hostfile requires the list of the servers. In both cases, there is no simple way to address a list of nodes as in a regular cluster, where machines are often named according to a defined pattern (e.g. `node-`*X*). Instead, we need a discovery service to find out which IP addresses correspond to the instances in our network. Contrarily to (Nguyen and Bein, 2017), however, we decided to exploit the own naming service of Docker by making "low level" DNS calls using *dig*. By making specific queries for the name of a service, we obtain a list of the corresponding IP addresses of the instances. As the hostfile also indicates the number of process (or slots) a node can run simultaneously, we call the *nproc* application on each machine, obtaining therefore the number of available processing cores. This simple "hack" is presented below, where we obtain the list of all *worker* nodes (i.e., instances of the "worker" service on Swarm).

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

3.2 Roles and External Access

In addition to the list of nodes, MPI strongly relies on the nodes' *rank*. For instance, the most important node in an MPI execution is the one tagged with rank 0, who usually starts the MPI community and gathers the results at the end. As this "master" node has some more responsibilities than a regular "worker" node, it is important to allow users to access this node using

SSH, for example. Indeed, several applications require an access to a frontend node where the user can execute preprocessing steps, setup the application parameters or simply verify the code is running before deploying it over the cluster. Therefore, we looked for ways to launch the master together with the workers. As we need to publish the master service's port directly from the Swarm node, this node cannot simply use the ingress routing network, but needs to be executed under the special `global` deployment mode.

Additional attributes ensure that the master will be easily located (on the manager node from the Swarm cluster), simplifying the access (using SSH) and also guaranteeing that at least this node mounts correctly all external volumes required for the application. Listing 1 presents the main elements of the `docker-compose.yaml` file used to define and deploy the Swarm service for our application.

Listing 1: Excerpt of the Swarm Service definition.

```
version: "3.3"
services:
  master:
    image: XXXXX
    deploy:
      mode:
        global
      placement:
        constraints:
          - node.role == manager
    ports:
      - published: 2022
        target: 22
        mode: host
    volumes:
      - "./WPS_GEOG:/WPS_GEOG"
      - "./wrfinput:/wrfinput"
      - "./wrfoutput:/wrfoutput"
    networks:
      - wrfnet
  workers:
    image: XXXXX
    deploy:
      replicas: 2
      placement:
        preferences:
          - spread: node.labels.datacenter
    volumes:
      - "./WPS_GEOG:/WPS_GEOG"
      - "./wrfinput:/wrfinput"
      - "./wrfoutput:/wrfoutput"
    networks:
      - wrfnet
networks:
  wrfnet:
    driver: overlay
    attachable: true
```

4 THE WRF MODEL

In order to experiment our virtualized cluster platform, we adapted the Weather Research and Forecasting (WRF) model (Skamarock et al., 2008), a well-known numerical weather prediction model. WRF has over 1.5 million lines in C and Fortran, as well as many dependencies on external software packages for input/output (I/O), parallel communications, and data compression, that are not trivial to satisfy. Hence, compilation and execution can be challenging for beginners or for users that do not have administration rights on their computing infrastructures.

Running the model can also be difficult for new users. WRF is composed by several steps to generate computational grids, import initialization data, produce initial and boundary conditions, and run the model (Hacker et al., 2017).

The typical workflow to execute the WRF model (Figure 2) is made of 5 phases, as indicated below. These steps do not include the additional access to external data sources, neither the analysis/visualization of the results.

1. Geogrid - creates terrestrial data from static geographic data
2. Ungrib - unpacks GRIB meteorological data obtained from an external source and packs it into an intermediate file format
3. Metgrid - interpolates the meteorological data horizontally 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

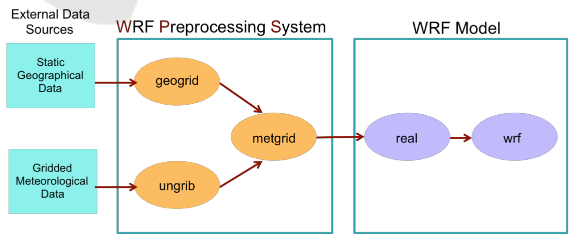

Figure 2: WRF workflow.

The three first steps are part of the WRF Preprocessing System (WPS), that is configured and compiled separately from the WRF Model. The WPS configuration allows two execution modes: `serial` or `dmpar` (distributed memory parallelism through MPI).

In the case of the WRF Model configuration, four modes are proposed: `serial`, `smpar` (shared memory parallelism), `dmpar` (distributed memory parallelism) and `sm+dmpar`. The `smpar` option depends on OpenMP, while the `dmpar` lies on MPI. The last option (`sm+dmpar`) combines OpenMP and MPI, but several works point out that the pure `dmpar` usually outperforms the mixed option (Council, 2010; Langkamp and Böhner, 2011).

Software containers, which are becoming an essential part of modern software development and deployment, offer a way for mitigating or eliminating many of the problems cited above, and simplify the deployment of computing infrastructures for both education and research. Containers allow the packaging of a working (and validated) WRF instance, ready to be used, preventing the user from having to install and to set up all dependencies.

4.1 WRF Containers for ARM

Although a non-official container for WRF on x86 platforms[5] developed by NCAR researchers exists already, this image is not adapted for cluster deployment and has not evolved since its launching, despite the intentions of the authors (Hacker et al., 2017).

When we started developing a version compatible with the SoCs under the ARM platform, we had to address a few issues related to the availability of some libraries and compiling options. Indeed, the original container image from NCAR is based on CentOS, which does not support ARM processors yet. This forced us to move to Ubuntu as our new base image. Not only Ubuntu supports ARM but most libraries required by WRF are available as packages, simplifying the installation (this reason also motivated us to avoid `alpine`, a popular image for containers).

The other issue is related to the pre-configuration of the parameters for WRF compiling. While WRF supports several compilers (gcc, Intel, Portland, etc.) and architectures, ARM processors are not listed among the supported ones. Fortunately, a few researchers have faced the same problem before[6] and we were able to apply their instructions. 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 modified the way input data is accessed, moving from a fixed Docker volume to a mounted file system. We believe that 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 a storage problem that may touch many SoC boards. Indeed, the first step on the WRF workflow (`Geogrid`) depends on a large geographical database (`WPS_GEOG`). Without careful pruning, the full database reaches 60GB when uncompressed, which is too voluminous for most SoC boards. By allowing the use of external volumes, we allow the users to attach external storage drives to their nodes. As demonstrated later in Section 5.2, only a single node requires this database, so we can minimize the costs and the management complexity in the SoC cluster.

Finally, we also updated the WRF version to 3.9.1.1, as the version present on the NCAR image dates back to 2015. As WRF 4.0 has recently been launched, we are planning to develop new images for this version.

As a result, WRF containers for both ARM and x86 architectures are now available at Docker Hub [7] and the scripts and Docker files are available at GitHub [8] .

[5]https://github.com/NCAR/container-wrf

[6]http://supersmith.com/site/ARM.html

5 PERFORMANCE BENCHMARKING

In order to assess the interest of using SoC based on ARM for meteorological simulations with WRF, we conducted a series of benchmarks to evaluate the performance of the application. The next sessions describe the experiments and the platforms we compared.

5.1 Definitions

For the benchmarks we used a dataset for a 12-hour forecasting on October 18, 2016 and concerning an area covering Uruguay and the south of Brazil. Although small, this dataset is often used as training example for meteorology students at Universidade Federal de Santa Maria, who can modify the parameters and compare the results to the ground truth observations. The entire dataset is accessible at our github repository.

In the benchmarks we compared different SoC models and a x86 computers. The SoC boards include a **Raspberry Pi 2 model B** (Broadcom BCM2835 processor, ARM Cortex-A7, 4 cores, 900MHz, 1GB RAM) and two **Raspberry Pi 3** (Broadcom BCM2837 processor, ARM Cortex-A53, 4 cores, 1.2GHz, 1GB RAM). The x86 computers

[7]https://hub.docker.com/r/lsteffenel/wrf-container-armv7l/

[8]https://github.com/lsteffenel/wrf-container-armv7l-RaspberryPi

were represented by a server with an Intel Xeon E5-2620v2 processor (2.10 GHz, 12 cores, 48GB RAM). We also experimented with other SoC boards like NanoPi NEO (Allwinner H3, ARM Cortex-A7, 4 cores, 1.2GHz, 512MB RAM), an NTC C.H.I.P. (AllWinner R8 processor, ARM Cortex-A8, 1 core, 1GHz, 512MB RAM) and a Banana Pi (Allwinner A83T processor, Arm Cortex-A7 8 cores, 1.8 GHz, 1GB RAM), but their poor performances or incompatibilities with Docker forced us to exclude these platforms from the subsequent tests.

All measures presented in this section correspond to the average of at least 5 runs. For the Docker Swarm clusters, we interconnected the devices via a 1 Gbps switch over RJ45, to avoid unreliable results due to the wireless connections.

Furthermore, as the WRF workflow is composed by 5 steps, we computed the execution time of each step individually, in order to assess the best deployment strategy. Therefore, the next sections present the separate analysis of the preprocessing steps (WPS+real) and the forecast step (WRF).

5.2 WPS and Real

As explained in Section 4.1, the size of the geographical database used by the Geogrid step on WPS often poses a problem for typical SoC internal storage. Indeed, in our experiments, we had to attach an external USB storage device to a Raspberry Pi node to accommodate the WPS_GEOG database.

Because WPS can be compiled with the `dmpar` option, we first tried 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).

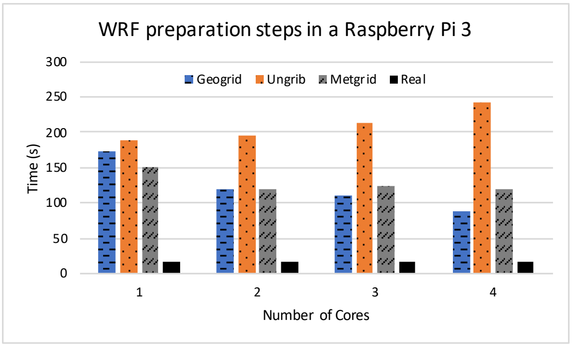

Figure 3: Performance of WPS steps when varying the number of cores.

The result of this benchmark, illustrated in Figure 3 and detailed in Table 1, indicates that only the *Geogrid* step effectively benefits from a multi-core execution. In the case of *Ungrib*, the parallel execution even penalizes the algorithm. 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. Additional benchmarks on the network performance, such as those conducted by (Yong et al., 2018), may also help tuning the different steps.

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

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

Even if *Geogrid* presents some performance improvements when run in parallel, 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 Section 5.3), 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.

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.

5.3 WRF

Contrarily to the preprocessing steps that finally represent only a small computing load, the WRF forecast is the main workload of the workflow. This is even more important on "production" environments, where more than a simple 12-hour forecast need to be computed.

Indeed, the forecasting step of WRF can benefit from multicore and cluster scenarios. Figure 4 indicates the average execution time when executing the WRF step on a single Raspberry Pi 3 (1 to 4 cores), on a cluster with two Raspberry Pi 3 (summing up 8 cores) and on a Swarm cluster with two Raspberry Pi 3 and one Raspberry Pi 2 (summing up 12 cores).

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

Cores	R Pi 2	R Pi 3	Xeon
1	6268.96	5647.47	539.05
2	3280.34	2473.89	314.69
3	2468.89	1801.18	264.53
4	2075.88	1602.68	173.55

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

Machines	Cores	Pi Swarm
1 x Pi 3	4	1602.68
2 x Pi 3	8	1322.42
2 x Pi 3 + Pi 2	12	1306.10

If multicore execution allows an important performance gain, the Swarm cluster executions show more mitigated results. As WRF is regularly executed in production clusters with the `dmpar` mode (MPI) and that the Docker overlay network imposes little overload, we suspect that the reduced performance gain is related to the network performance on the Raspberry Pis. Indeed, as observed by (Beserra et al., 2017), the access to the communication bus is a recurrent problem on SoC, and the Raspberry Pi suffer from a “low” speed interconnection card (10/100 Mbps only).

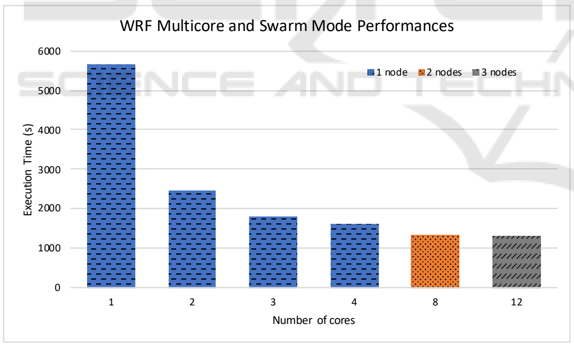

Figure 4: Performance of WRF in multicore and swarm cluster mode.

Tables 2 and 3 detail these results, and also present a performance comparison with a Xeon processors. While the x86 processors are faster, the execution time on the Raspberry Pis is still acceptable, enough to deliver forecasts on a daily or even hourly basis, or for education and training. If we consider the material and environmental cost of the SoC solution, it is indeed an interesting alternative for scientific applications like WRF.

6 CONCLUSIONS

This work focuses on the deployment of containerized scientific applications over a cluster of SoC-based systems. Most System-on-a-Chip (SoCs) are based on the ARM architecture, a flexible and well-known family of processors that now started to infiltrate the HPC (High Performance Computing) domain. Container-based virtualization, on the other side, enables the packaging of complex applications and their seamless deployment. Together, SoC and containers represent a promising alternative for the development of computing infrastructures, associating the low cost and minimal maintenance of SoCs and the flexibility of containers.

Nonetheless, most traditional scientific applications rely on MPI for scalability, and popular container managers like Docker do not offer a proper support for MPI. We therefore propose, in a first moment, a service specification to deploy a Docker Swarm cluster that is ready for MPI applications. Later, we study how to adapt the WRF meteorological forecast model to run on ARM-based SoCs. Benchmarks on different SoC platforms are used to evaluate the performance and the interest of using containers over SoC clusters. These results indicate that if popular 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 development of a generic platform capable of accommodating other MPI applications, as well as the support for recent versions of WRF and its integration on more elaborated frameworks.

ACKNOWLEDGEMENTS

This research has been partially supported by the French-Brazilian CAPES-COFECUB MESO project [9] and the GREEN-CLOUD project [10] (#16/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

REFERENCES

Ali, M., Vlaskamp, J. H. A., Eddin, N. N., Falconer, B., and Oram, C. (2013). Technical development and socioeconomic implications of the Raspberry Pi as a learning tool in developing countries. In *Computer Sci-*

[9] http://meso.univ-reims.fr
[10] http://www.inf.ufrgs.br/greencloud/

ence and Electronic Engineering Conf. (CEEC), pages 103–108. IEEE.

Alvarez, L., Ayguade, E., and Mantovani, F. (2018). Teaching HPC systems and parallel programming with small-scale clusters. In *2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, pages 1–10.

Azab, A. (2017). Enabling Docker containers for high-performance and many-task computing. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 279–285.

Beserra, D., Pinheiro, M. K., Souveyet, C., Steffenel, L. A., and Moreno, E. D. (2017). 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)*, pages 363–370.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57.

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

Council, H. A. (2010). Weather research and forecasting (WRF): Performance benchmark and profiling, best practices of the HPC advisory council. Technical report, HPC Advisory Council, http://www.hpcadvisorycouncil.com/pdf/WRF_Analysis_and_Profiling_Intel.pdf.

Cox, S. J., Cox, J. T., Boardman, R. P., Johnston, S. J., Scott, M., and O'brien, N. S. (2014). Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, 17(2):349–358.

d. Bayser, M. and Cerqueira, R. (2017). Integrating mpi with docker for hpc. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 259–265.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2014). An updated performance comparison of virtual machines and linux containers. *IBM technical report RC25482 (AUS1407-001), Computer Science*.

Hacker, J. P., Exby, J., Gill, D., Jimenez, I., Maltzahn, C., See, T., Mullendore, G., and Fossell, K. (2017). A containerized mesoscale model and analysis toolkit to accelerate classroom learning, collaborative research, and uncertainty quantification. *Bulletin of the American Meteorological Society*, 98(6):1129–1138.

Higgins, J., Holmes, V., and Venters, C. (2015). Orchestrating docker containers in the HPC environment. In Kunkel, J. M. and Ludwig, T., editors, *High Performance Computing*, pages 506–513, Cham. Springer International Publishing.

Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20.

Langkamp, T. and Böhner, J. (2011). Influence of the compiler on multi-CPU performance of WRFv3. *Geoscientific Model Development*, 4(3):611–623.

Marathe, A., Harris, R., Lowenthal, D., de Supinski, B. R., Rountree, B., and Schulz, M. (2014). Exploiting redundancy for cost-effective, time-constrained execution of HPC applications on Amazon EC2. In *23rd Int. Symposium on High-Performance Parallel and Distributed Computing*, pages 279–290. ACM.

Molano, J. I. R., Betancourt, D., and Gómez, G. (2015). Internet of things: A prototype architecture using a Raspberry Pi. In *Knowledge Management in Organizations*, pages 618–631. Springer.

Montella, R., Giunta, G., and Laccetti, G. (2014). Virtualizing high-end GPGPUs on ARM clusters for the next generation of high performance cloud computing. *Cluster computing*, 17(1):139–152.

Morabito, R., Kjallman, J., and Komu, M. (2015). Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), IEEE Int. Conf. on*, pages 386–393. IEEE.

Nguyen, N. and Bein, D. (2017). Distributed MPI cluster with Docker Swarm mode. In *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–7.

Ruiz, C., Jeanvoine, E., and Nussbaum, L. (2015). Performance evaluation of containers for HPC. In Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M. E., Scarano, V., Varbanescu, A. L., Scott, S. L., Lankes, S., Weidendorfer, J., and Alexander, M., editors, *Euro-Par 2015: Parallel Processing Workshops*, pages 813–824, Cham. Springer International Publishing.

Skamarock, W. C., Klemp, J. B., Dudhia, J., Gill, D. O., Barker, D. M., Duda, M. G., Huang, X.-Y., Wang, W., and Powers, J. G. (2008). A description of the advanced research WRF version 3, NCAR technical note. *National Center for Atmospheric Research, Boulder, Colorado, USA*.

Somasundaram, T. S. and Govindarajan, K. (2014). CLOUDRB: A framework for scheduling and managing high-performance computing (HPC) applications in science cloud. *Future Generation Computer Systems*, 34:47–65.

Steffenel, L. and Kirsch-Pinheiro, M. (2015). When the cloud goes pervasive: approaches for IoT PaaS on a mobiquitous world. In *EAI International Conference on Cloud, Networking for IoT systems (CN4IoT 2015)*, Rome, Italy.

Weloli, J. W., Bilavarn, S., Vries, M. D., Derradji, S., and Belleudy, C. (2017). Efficiency modeling and exploration of 64-bit ARM compute nodes for exascale. *Microprocessors and Microsystems*, 53:68 – 80.

Wolf, W., Jerraya, A. A., and Martin, G. (2008). Multiprocessor system-on-chip (MPSoC) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713.

Yong, C., Lee, G.-W., and Huh, E.-N. (2018). Proposal of container-based hpc structures and performance analysis. 14.

Younge, A. J., Henschel, R., Brown, J. T., von Laszewski, G., Qiu, J., and Fox, G. C. (2011). Analysis of virtualization technologies for high performance computing environments. In *IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 9–16, Washington, DC, USA. IEEE Computer Society.