Integrating MPI with docker for HPC

Maximilien de Bayser
IBM Research
Rio de Janeiro
Email: mbayser@br.ibm.com

Renato Cerqueira
IBM Research
Rio de Janeiro
Email: rcerq@br.ibm.com

***Abstract*—Container technology has the potential to considerably simplify the management of the software stack of High Performance Computing (HPC) clusters. However, poor integration with established HPC technologies is still preventing users and administrators to reap the benefits of containers. Message Passing Interface (`MPI`) is a pervasive technology used to run scientific software, often written in `Fortran` and `C/C++`, that presents challenges for effective integration with containers. This work shows how an existing `MPI` implementation can be extended to improve this integration.**

I. Introduction

One of the most complex and laborious tasks in the administration of computing infrastructures is the management of the software stack. And when everything is running smoothly, the administrators are often reluctant to change it to accommodate new applications or new requirements of existing applications. This situation instigated the creation of the `DevOps` [1] methodology that among other things recommends the automation of infrastructure tasks to enable reproducible running environments. The practice of materializing this automation as executable scripts is called `Infrastructure as Code` (`IaC`) and when combined with version control tools such as `git` offers many advantages over manual configuration such as repeatable set-ups, documentation (in the form of code) and tracking of changes.

Container technology in the form of `Docker` containers [2] was rapidly adopted by the community of `DevOps` practitioners because it allows friction-less deployment of applications on the same server even with conflicting dependencies, but with much less resource consumption compared to virtual machines. `Docker` also follows the `IaC` approach: containers are instantiated from images which are specified using a script called the "`Dockerfile`". An image contains all software that is needed to run an application, it even includes a `Linux` [3] distribution. Once an image is built it can be deployed on any compatible `Linux` machine.

However, it is still difficult to reap the benefits of `Docker` containers for traditional High Performance Computing (`HPC`) clusters. `HPC` clusters usually have a central administration; users wishing to run applications have two options: asking an administrator to install the required software or making a custom installation by themselves in their home directory. Making custom installations often involves configuring and compiling software, so many users of `HPC` clusters without a computer science background choose the first option. This presents a challenge for the administrator because if he installs everything the users request, at some point he will likely run into a situation where two users require two different versions of a same library that cannot be installed together. A narrative of this kind of problem from a user perspective can be found in chapter 9 of the `Handbook of Research on Computational Science` [4]

To mitigate this problem, many cluster set-ups have a system of loadable modules [5]. These modules are custom installations of several versions of compilers and libraries, each in their own isolated directory tree, that by default are not included in the system path but can be loaded with a command. Although modules do help, there will always be some library that is not yet installed. And once there is a lot of software installed it is difficult to identify which software is still being used and what impact changes in the software stack could have. This is only aggravated by custom installations that don't have a scheme of explicit dependencies like software package managers offer.

Containers such as those implemented by `Docker` are an obvious solution for this problem. Inside a container the user can install all the software he needs using standard tools like package managers. He can choose the `Linux` distribution that he prefers and apply many system configurations. Once ready, the image could be deployed to the cluster without requiring any interference from the administrators.

One obstacle for the widespread adoption of `Docker` containers in `HPC` clusters is the difficulty of integrating it with the Message Passing Interface (`MPI`) framework [6]. `MPI` allows the parallel execution of programs by running an instance on each node in the cluster and taking care of the communication between them. The integration difficulty lies in the set-up phase where `MPI` instantiates a copy of the program on each machine, as we will be discussed in section II.

In the remainder of this work we will analyze these issues in depth and discuss solutions and weigh their benefits. Section II gives a brief introduction on `MPI` and `MPICH` and identifies the source of integration problems. Section III introduces `Docker` and `Docker Swarm`. Section IV presents a brief analysis of prior work and presents our solution. Section V analysis the strengths and weaknesses of this solution and discusses this solution in a broader context. Finally VI concludes this presentation.

II. MPI

`MPI` is an industry standard that defines a library interface for the message-passing parallel programming model. Much like the standard C library specification, it defines an abstract API that allows many independent implementations that are compatible with each other, thus a program written for `MPI`

978-1-5090-5817-4/17 $31.00 © 2017 IEEE
DOI 10.1109/IC2E.2017.40

is portable from one implementation to the other. MPI also specifies a set of command line programs that are used to start parallel executions or compile MPI programs.

MPI automatically starts as many copies of a program as the users requests and tries to allocate them on the available computing nodes automatically or according to user specification. Each program receives a unique numeric ID called a “rank”. By convention the instance with the rank 0 is often used to run initial set-up tasks or other coordination task. Once running, any instance can send a message to any other instance identified by rank or even groups of ranks. From the library’s point of view, the message is just an array of bytes, the semantic is defined by the application. Initially the API functions for sending and receiving messages were all synchronous, which means that the call to MPI_Send would only return once the receiving party had received the message. Conversely, the call to MPI_Recv would only return once a message was received. However after a few iterations of the standard, asynchronous calls were included. A small example of code taken from MPICH’s documentation can be seen in listing II.1

```
#include "mpi.h"
int main( int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD,
    &myrank );
  if (myrank == 0)
  /* code for process zero */
  {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1,
      MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1)
  /* code for process one */
  {
    MPI_Recv(message, 20, MPI_CHAR, 0,
      99, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

Listing II.1: Small example of MPI code

Because of the portability and the availability of bindings to C/C++ and Fortran, MPI was quickly adopted as the *de facto* standard to run numeric software on parallel computing architectures. As a consequence a large legacy of MPI software has accumulated over the years and still remains relevant. MPI versions of standard linear algebra libraries like BLAS and LAPACK were written to split operations on large matrices over several machines, as well as Finite Element solvers for mechanical simulation and many other numerical libraries. All this software infrastructure provides the backbone to a lot of software that is still extremely relevant today like weather forecast simulations, aerodynamics simulations for new airplanes or geomechanical simulation of petroleum reservoirs. Basically, MPI programs have been driving the impressive increase in FLOPS in the TOP500 ranking of supercomputers [7].

To run an MPI program, the user calls mpiexec on the command line passing as arguments a specification of the number of instances and nodes and an absolute or relative path to the executable file of the program, followed in turn by command line arguments specific to the program. The MPI runtime then tries to allocate these processes on the available computing resources. Typical computing clusters consist of one or more login servers were users log in and run commands, one or more file system servers and other auxiliary servers and, most importantly a large array of identical worker machines. The prevalent operating system on these clusters is some flavor of UNIX and the MPI implementation takes advantage of standard UNIX tools like ssh (Secure Shell) to access worker nodes and start processes on them. Therefore, the typical work flow is the following: (1) the user call mpiexec; (2) mpiexec uses ssh to start an auxiliary program on each requested computing node; (3) the auxiliary program connects back to the main program and runs the user process setting up the communication infrastructure, assigning it a rank; (4) the auxiliary program waits for the user program to exit and forwards everything that is written to the standard output to the main program.

This work flow that is internal to MPI is exactly where the calls to Docker should happen. This is the reason for the integration difficulties because ideally this is where the containers should be allocated on demand. Another minor difficulty is that the MPI runtime assumes that the path to the executable, as well as the current working directory of the master process is valid on each node. In other words, the file system on every node must be identical with respect to these paths or be a shared file system. For usability reasons usually a shared file system such as NFS, GPFS or Lustre is used.

A. MPICH

MPICH [8] is an open-source implementation of MPI. It was initially developed by the Argonne National Laboratory during the standardization process of MPI to provide feedback about implementation and usability issues. It is still actively developed and is used as foundation for many commercial implementations of MPI.

In MPICH, program setup is handled by a launching framework called Hydra. During execution it is composed by two logical parts, one represented by the mpiexec program that runs on the master node and one represented by the hydra_pmi_proxy program, which runs on each worker node. Hydra internally has a modular architecture of resource management kernels (RMK) and launchers (see figure 1, taken from the MPICH wiki). The resource management kernels are used to integrate with cluster resource management software such as SLURM [9] or LSF. The launchers are used to launch the hydra_pmi_proxy on computing nodes. The default launchers uses ssh to accomplish this. All launchers start the proxy program passing as command line argument the host

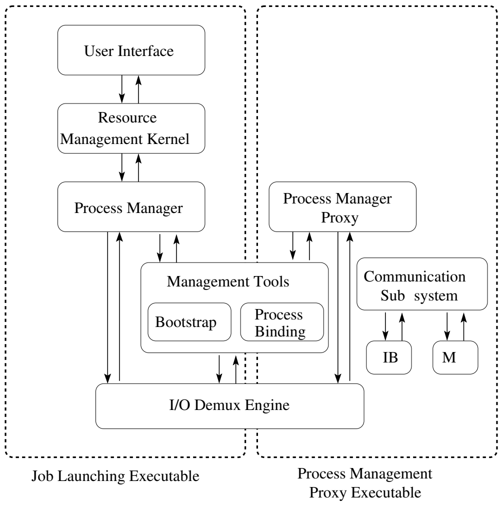

Fig. 1. Hydra's architecture

and port of the mpiexec program and the command line for the MPI program.

Although the architecture of launches and RMKs is modular, in the way the build process is structured, all available implementations must be compiled-in. Therefore, to extend this framework one has to compile new implementations together with Hydra's source code or implement a plug-in architecture to load launchers and RMKs at runtime. For simplicity, we chose the first alternative for this work.

III. Docker and Docker Swarm

Docker is an abstraction layer on top of several isolation features provided by the Linux kernel, like process name space isolation and file-system isolation. An isolated environment where one or more processes run is called a container. Processes running in different containers a priori share nothing but the Linux kernel. These container management features are provided to the user as a single unified command line interface.

Docker also provides an ingenious system of container templates, called images. An image is a package containing all the directory structure and files that must be present in the container. Most importantly, images are layered. When the user creates a new image, he can use an existing image as base, and all the changes that he introduces to this file system are recorded. Therefore, instead of keeping a copy of the old file system and a copy of the new file system, only the old file system and the changes are stored, reducing storage requirements, much like a source code version control software would. The base file system and the difference file systems are called layers and are unified as a single file system by specialized kernel modules. This kind of file system is called union file system in Linux terminology.

When a container is run, instead of copying the entire image, another layer is added to the top of the stack. All changes made by a container are restricted to this top layer. For instance if an 1GB image contains software that does not modify the file system, one can instantiate a thousand containers simultaneously without using more that the initial 1GB. This represents a massive saving of disk space compared to virtual machines.

Docker also allows a container to access file systems outside of the union file system. These are called volumes in Docker terminology and are attached to specific points in the union file system much like traditional UNIX mount points. Volumes, can be anonymous, named or be tied to a specific host path chosen by the user. The latter form can be used to give a container access to portions of the host file system.

Regarding networking, the default configuration is for containers to have their own IP address in a virtual bridged network. With a command line parameter the user can ask Docker to forward host network ports to container network ports. One can also create a virtual network where containers can be added. In this network containers can reach each other using virtual host names or IP addresses.

Architecturally, Docker is divided in two main components: a command line user interface program called docker and a background process that executes with administrative privileges that actually manages the containers. This background process is called the docker *daemon* after the UNIX terminology.

At first sight, containers may look like a more economical replacement for virtual machines. However, containers lack a separate operating system and process management and should be seen more as a software package that includes everything but the kernel.

Docker containers can be a good match for HPC requirements because unlike virtual machines the computing resource overhead is low and there is no hardware emulation. A container can access specialized hardware resources like GPUs,FPGAs, infiniband network interfaces just like any other process on the same machine.

A. Docker Swarm

When running Docker containers on a cluster, the user usually does not care on which specific worker machine his containers run. The machines are just a pool of computing resources to him. And he should not make the decision of which computing node to use because chances are that he would make a sub-optimal choice. What the user really needs is a system that manages a cluster of Docker hosts for him.

There are today basically a few choices of container cluster managers: kubernetes [10], Marathon [11], Fleet [12], and Docker Swarm. Docker Swarm is developed by the same organization that develops Docker and therefore integrates well with it. The configuration is also quite simple and it has the advantage that the same docker command line tool can be used to submit containers. For these reasons we have chosen to use it for our experiments.

Docker Swarm consists of one or more master servers on the master machines and a collection of slave servers on the worker machines. They communicate using service discovery servers like consul or etcd. The servers are usually installed as containers themselves simplifying the installation.

When Docker Swarm is installed, an environment variable must be set so that the Docker client connects to the swarm server instead of to the local Docker *daemon*. When the user runs a new container, the swarm manager selects a node and forwards the request to the local Docker *daemon*. A convenient feature is that virtual networks created by Docker swarm work across nodes. Volumes however are local to each node, but since cluster usually have shared file systems this is not a problem.

Since it is transparent to the user of the Docker client if he is using a local Docker *daemon* or a Docker Swarm, an integration with MPI can take advantage of that to work in both kinds of environments.

IV. Integrating docker with MPI

Out of the box there is no simple and satisfying way to integrate Docker with MPI programs. A first approach is to instantiate a Docker container on each worker node with a running ssh server and then give mpiexec the list of the container host names. This seems to be the approach followed by Hsi-En Yu *et al.* [13] mpiexec should be run in a container as well and all containers should be attached to a private virtual network to take advantage of intra-container networking and network isolation. The problem with this approach is that it does not only go against the recommendation of not using a container like a virtual machine, but it is also wasteful in resources and presents a poor usability. When the virtual cluster of containers is not being used, it still occupies resources just to run the ssh servers. And for the layman it is cumbersome to start a container on each node including the master node and running mpiexec from within the master container. Also a cluster administrator probably would have trouble to convince the users to adopt this complicated scheme.

Of course, all of this could be automated, but the inclusion of ssh component would a source of accidental complexity. At first, a monitoring process would have to monitor the containers and wait for each one to be up and running. In addition, the monitoring process could be terminated prematurely, forcing the implementation signal handling scheme. And if the user or system administrator used the UNIX SIGKILL signal on the monitoring process the orphaned containers would have to be stopped and removed manually. And finally, this solution forces the installation of ssh in all images, adding a small storage and configuration overhead. As explained below, these issues can be avoided by tying the lifetime of the container to the duration of the MPI task.

A. The Proposed Solution

The approach that we followed in this work takes advantage of a special feature of the MPICH implementation. We have chosen to use MPICH, version 3.2, as described in section II-A, because of its flexible process management framework, Hydra. As mentioned in section II, the critical point where MPI should interact with Docker is when it launches the initial bootstrap program on the allocated nodes. Hydra has a special manual launcher that does nothing but print on the standard output the parameters that it would have started the bootstrap program with, including the port where the master process is waiting for connections from the bootstrap processes.

We have developed a wrapper program that the user calls instead of mpiexec. This program takes all command line arguments that mpiexec takes and forwards those to mpiexec. It also expects an extra command line argument detailing the Docker image that the user would like to run.

In detail, this wrapper:

1) creates a docker *overlay network*;
2) parses local MPI configuration files and command line parameters to find out how many hosts should be allocated
3) creates a container from the user-specified image in this virtual network and runs mpiexec in it, adding the -launcher manual argument to specify the initial bootstrap method
4) for each proxy invocation printed by the manual launcher, starts a new container in the same virtual network, running hydra_pmi_proxy
5) forwards all standard output of the master program to the use
6) waits for the master container to exit
7) deletes the overlay network.

An overlay network is created because every container in this network can freely connect to any other container in the network, but no container outside. The network is created with a unique name, based on the wrapper process PID to avoid naming collisions.

A critical point is the specification of nodes: in a normal mpiexec invocation the computing nodes that the users wish to use to run his computation are supplied to mpiexec via command line parameters or default configuration files. However, since the MPI program is run inside of containers, instead of hosts, passing these host names to mpiexec would not be correct. Therefore we use the host information only to find out how many host the user expects to use and start as many worker containers. Each of those containers is assigned a name, and this new list of names is passed to mpiexec instead of the old one.

In this approach, all load balancing is left to Docker Swarm.

Another possible approach of integration is to eliminate the need for a Docker Swarm installation. The wrapper could use ssh to start containers on computing nodes. In this case the host names passed by mpiexec to hydra_pmi_proxy would again have their original semantic. On one hand this would also allow for a deterministic placing of containers, but on the other hand it would forego the benefits of transparent container placing.

In the beginning we also considered adding a new launcher to MPICH. However, since the master process also has to be inside a container in the overlay network, we would have to give this launcher access to the Docker *daemon* from inside the container. This would introduces unnecessary complexity and security risks.

The wrapper by default mounts the user's home directory in each container. Since it usually is a shared file system, all containers have access to the same files. The wrapper also runs the main process in each container with the user's UID. This improves the security and prevents file permission problems.

The user can also pass extra Docker command line arguments to the wrapper, but these are subject to permission checks, as will be discussed in section IV-B

To use the Docker launcher the user needs to wrap his software inside an image that has a MPICH installation. In listing IV.1, a Dockerfile for an image based on the Fedora distribution is showed.

```
FROM fedora:24
RUN dnf install -y mpich-devel
```

Listing IV.1: A sample Dockerfile

The wrapper could also call the Docker *daemon* directly over its API instead of calling the command line tool. Technically this would be equivalent, therefore for the sake of simplicity we chose to invoke the command line that is already familiar to us.

B. Security

In many systems the access to the Docker *daemon* is restricted, since Docker can effectively be used to get root access on the host. The wrapper does allow to pass extra command line arguments to the Docker invocations that create the container. However, by default all extra arguments are forbidden and the administrator must white list the options he thinks are safe. These permissions are configured in a system wide configuration file where the administrator can also set mandatory command-line arguments to, for example, always mount a specific directory or drop Docker privileges.

In order to allow the wrapper program to invoke docker, the system administrator can add the UNIX setuid permission to make the wrapper process run with higher privileges. Otherwise the docker client will run with the user's UID and might be prohibited from connecting to the Docker *daemon*. Considering the inherent risk that setuid executables present, the wrapper program is written in a programming language that does not allow unsafe memory operation, a major source of vulnerabilities and privilege escalation exploits. In any case, it is recommended to run SELinux or AppArmor to tighten the security.

We could also make the wrapper call the Docker *daemon* directly over HTTP or UNIX socket. Over HTTP our wrapper would need special privileges as well because it would have to present a valid certificate to identify itself to the *daemon*. And this certificate should be stored in a way that is inaccessible to the users, but accessible to this wrapper they are calling. As for the UNIX socket, the permissions to open the socket are the same as the permission needed to call the docker command.

This wrapper can also be used to improve the security of sensitive data. When creating an overlay network, one can ask for encrypted network connections, so effectively, all MPI traffic is encrypted.

V. Discussion

The one who benefits the most from this work is perhaps the system administrator since he can delegate almost all software administration to the users themselves. But for the user there are also several advantages. The most important one is probably the reduction of bureaucracy, since he no longer needs to ask the administrator to install software for him. The second benefit in importance is reproducible deployments and the ability to share Dockerfiles with his colleagues. This file also serves as a precise documentation of the container configuration and if kept under version control also allows him to go back in time and run older versions of his experiments.

A drawback of this solution is that users have to familiarize themselves with Docker and also have to use a slightly longer command line. An example of an original command line is showed in listing V.1 and a modified command line in listing V.2. An administrator could, however provide a simple wrapper script for his users to simplify usage.

```
mpiexec -n 40 myprogram arg1 arg2
```

Listing V.1: Standard MPI invocation

```
dockermpi mpi_image -n 40 myprogram arg1 arg2
```

Listing V.2: Modified MPI invocation

In terms of resource usage, containers are slightly more wasteful of disk space than native installation since they contain redundant copies of basic system libraries and tools. This is aggravated with Docker Swarm because images must be copied to every node. As of the time of writing, Docker does not support sharing a single image repository among several Docker *daemons*.

In the remainder of this section we discuss several other consequences of this work.

A. High-speed networks

The default overlay network driver uses network bridges to create a virtual network, but in a HPC environment it would be preferable to use existing high-speed networking fabric like Infiniband directly. We still have to run more tests to explore this possibility.

B. Integration with other cluster managers

Most HPC clusters have a cluster management system that provides a job queue were users submit their jobs. The system guarantees that each user gets his fair share of computing resources and that only as much jobs as the machine supports are run simultaneously. Some examples of cluster management systems that MPICH support are SLURM, SGE and LSF.

Section IV discussed an alternative implementation that foregoes the need for Docker swarm. Such an implementation could check the availability of computing nodes with a cluster management system first to decide where to instantiate the worker containers. This is not yet possible with Docker swarm because it does not allow to influence the placement of containers.

C. MPI with docker in the cloud

Cloud Computing enables on-demand network access to a shared pool of configurable computing resources that should be rapidly provisioned and released with minimal management effort or service provider interaction [14] [15].

With the widespread adoption of `Docker`, many cloud providers have started to offer the possibility to run containers in the cloud. `CloudFoundry` is an example of such a cloud platform. To run containers the user can run containers using a `CloudFoundry`-specific command line tool or instruct the `Docker` command line tool to access the remote API endpoint that implements `Docker`'s API.

This transparent use of different kinds of computing infrastructure using the same `Docker` commands then allows us to use the same `MPICH Docker` wrapper to run `MPI` jobs in the cloud. In the future this could also enable the offloading of computing to the public cloud when local computing resources are exhausted as proposed by Kim *et al.* [16], Although this would require more sophisticated mechanisms to be included in the launcher.

A good integration with cloud platforms could help realize the vision of virtual clusters on demand [17] [18] [19], since

D. Benefits to e-Science

Science advances as scientists build on the previous achievements of their peers and for that it is often required to first independently reproduce their results. However, today many advances in science are driven by the use of computing. New insights in biology, astronomy, chemistry and many other fields are the result of complex calculations executed by sophisticated computing tools. Recognizing this, many researchers are publishing their code in online repositories to help others [20]. The code, however is only a piece of what is necessary to reproduce the results of other [21] [22] [23] [24]. Much like in the deployment of web-services, an uniform environment configuration is critical. If researchers would also publish the `Dockerfile` they used to build their containers it could greatly reduce the trouble of other researchers. Carl Boettinger lists four primary issues that researchers face: (1) "dependency hell", (2) "imprecise documentation", (3) "Code rot", (4) Lack of computer science training [25]. He argues that `Docker` containers could significantly impact the first three issues. He does not however mention the issue with `MPI` compatibility.

E. Related work

Several pages on the internet describe how to set up a cluster of `ssh`-enabled containers as described in section IV ([26]) and there is even explicit support for this integration approach on a commercial cloud service ([27]). However, in environment with many users, unless overlay networks are used, this leads to the problem of allocating ports for the `ssh` *daemons* to listen on. It also leaves idle containers running after the `MPI` task has finished that must be terminated explicitly. Christian Kniep describes a proof of concept similar to ours, based on the `OpenMPI` implementation, but it appears that there are no further developments [28]. `Singularity` is an alternative container technology developed by the Lawrence Berkeley National Laboratory [29]. It is designed for `HPC` and avoids several issues such as the high privilege needed to run `Docker` containers. And finally there are several solutions to run single containers with existing cluster management software ([30]).

VI. Conclusion

In this paper we have identified the source of the integration problems between `Docker` and `MPI` and discussed the impacts of such an integration both for cluster administrators and the e-Science community. We have discussed how this integration could bring to HPC the same benefits that container technology has already brought to other kinds of software deployment. We have shown how to achieve this integration by creating a front-end that orchestrates the allocation of containers to run `MPI` processes. We believe that this approach is readily applicable to other `MPI` implementations as well. Other container cluster managers like `kubernetes` could be supported as well. The code of this Hydra extension can be found at https://github.com/maxdebayser/mpich-docker-integration

VII. Acknowledgements

This work has been partially supported by FINEP/MCTI under grant no. 03.14.0062.00

References

[1] M. Hüttermann, *DevOps for Developers*. Springer, 2012, vol. 1.

[2] "Docker." [Online]. Available: https://www.docker.com/

[3] L. Torvalds, "Linux [operating system]." [Online]. Available: https://github.com/torvalds/linux

[4] J. Segal and C. Morris, *Handbook of Research on Computational Science and Engineering: Theory and Practice*. USA: IGI Global, 2011, vol. 1, ch. Developing Software for a Scientific Community:Some Challenges and Solutions, pp. 177–196.

[5] J. Layton, "Environment modules a great tool for clusters." [Online]. Available: http://www.admin-magazine.com/HPC/Articles/Environment-Modules

[6] M. P. I. Forum, "Mpi: A message-passing interface standard," 2015. [Online]. Available: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[7] E. Strohmaier, "Top500 supercomputer," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188474

[8] "Mpich." [Online]. Available: https://www.mpich.org/

[9] A. B. Yoo, M. A. Jette, and M. Grondona, *SLURM: Simple Linux Utility for Resource Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. [Online]. Available: http://dx.doi.org/10.1007/10968987_3

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2890784

[11] Mesosphere, "Marathon." [Online]. Available: https://mesosphere.github.io/marathon/

[12] CoreOS, "Fleet." [Online]. Available: https://github.com/coreos/fleet

[13] H. Yu and W. Huang, "Building a virtual HPC cluster with auto scaling by the docker," *CoRR*, vol. abs/1509.08231, 2015. [Online]. Available: http://arxiv.org/abs/1509.08231

[14] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," NIST, Tech. Rep., 2011.

[15] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud computing: Principles and paradigms*. John Wiley & Sons, 2010, vol. 87.

[16] H. Kim, Y. el Khamra, S. Jha, and M. Parashar, "An autonomic approach to integrated hpc grid and cloud usage," in *Proceedings of the 2009 Fifth IEEE International Conference on e-Science*, ser. E-SCIENCE '09. IEEE Computer Society, 2009, pp. 366–373.

[17] R. Strijkers, W. Toorop, A. v. Hoof, P. Grosso, A. Belloum, D. Vasuining, C. d. Laat, and R. Meijer, "AMOS: Using the cloud for on-demand execution of e-science applications," in *Proceedings of the International Conference on e-Science*. IEEE Computer Society, 2010, pp. 331–338.

[18] S. Childs, B. Coghlan, and J. McCandless, "Gridbuilder: A tool for creating virtual grid testbeds," in *Proceedings of the International Conference on e-Science and Grid Computing*. IEEE Computer Society, 2006, pp. 77–.

[19] J. J. Rehr, K. Jorissen, F. D. Vila, and W. Johnson, "High-performance computing without commitment: Sc2it: A cloud computing interface that makes computational science available to non-specialists," in *Proceedings of the International Conference on E-Science*. IEEE Computer Society, 2012, pp. 1–6.

[20] N. Barnes, "Publish your computer code: it is good enough," *Nature*, no. 467, Oct. 2010.

[21] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Commun. ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/2812803

[22] J. T. Dudley and A. J. Butte, "In silico research in the era of cloud computing," *Nature biotechnology*, vol. 28, no. 3, pp. 1087–1185, Nov. 2010.

[23] E. Eide, "Toward replayable research in networking and systems," 2010.

[24] R. e. a. FitzJohn, "Reproducible research is still a challenge." 2014. [Online]. Available: http://ropensci.org/blog/2014/06/09/reproducibility

[25] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723882

[26] Z. M. Wang, "Developing message passing application (mpi) with docker." [Online]. Available: https://www.ibm.com/developerworks/community/blogs/W8932c25e7e86_409c_ab4c_b1deebf711ff/entry/developing_message_passing_application_mpi_with_docker?lang=en

[27] M. Azure, "Batch shipyard." [Online]. Available: https://azure.github.io/batch-shipyard/

[28] C. Kniep, "dssh: Proof of concept for a ssh-less, docker-native mpi." [Online]. Available: http://www.qnib.org/2016/03/31/dssh/

[29] L. B. N. Laboratory, "Singularity." [Online]. Available: http://singularity.lbl.gov/

[30] IBM, "Use ibm spectrum lsf with docker." [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSWRJV_10.1.0/lsf_welcome/lsf_kc_docker.html