10-22-2018

Containers and Reproducibility in Scientific Research

Sara Faraji Jalal Apostal

David Apostal

Ronald Marsh
University of North Dakota, ronald.marsh@UND.edu

Containers and Reproducibility in Scientific Research

Sara Faraji Jalal Apostal, David Apostal, Ronald Marsh
Department of Computer Science University of North Dakota
sara.farajijalal@und.edu, david.apostal@und.edu, ronald.marsh@engr.und.edu

Abstract—Numerical reproducibility has received increased emphasis in the scientific community. One reason that makes scientific research difficult to repeat is that different computing platforms calculate mathematical operations differently. Software containers have been shown to improve reproducibility in some instances and provide a convenient way to deploy applications in a variety of computing environments. However, there are software patterns or idioms that produce inconsistent results because mathematical operations are performed in different orders in different environments resulting in reproducibility errors. The performance of software in containers and the performance of software that improves numeric reproducibility may be of concern for some scientists. An existing algorithm for reproducible sum reduction was implemented, the runtime performance of this implementation was found to be between 0.3x and 0.5x the speed of the non-reproducible sum reduction. Finally, to evaluate the impact of using a container on performance, the runtime performance of the WRF (Weather Research Forecasting) package was tested and found to be 0.98x of the performance in a native Linux environment.

Key Words — Software Containers, Numerical Reproducibility, Runtime Performance

I. INTRODUCTION

Scientists use complex applications and big data on High Performance Computing (HPC) systems to study and analyze some of the biggest questions in science today, such as what the Universe is made of and how it has evolved over time. With "traditional" single-core CPUs, documenting a numerical result was relatively straightforward. However, recent technology developments have made it difficult to produce consistent or reproducible results across different platforms. The predictability of the results is complicated when the systems grow in scale. Therefore, this situation affects the ability to rely on scientific computations.

Implementation of experimental learning and research with numerical application codes is challenging for users both in research and education. Most of the time running a scientific application is not easy for individuals. Specific steps to configure and compile the source code, set specific runtime options, import data sets, and so on are required. Frequent updates to the application code, operating systems and compilers make it difficult for users to compile and execute their software. We can put together all these steps into one or more scripts within a container, so that users with less knowledge about a complex application are able to run the application only by executing the script. In these cases, containers play an important role in the development and deployment of scientific applications because they help to eliminate the above-mentioned problems for users. For example, the user can deploy an application without any knowledge about the infrastructure, platform or underlying operating system [2].

Containers are a way to use a lightweight virtualization for executing several isolated Linux systems within a single host operating system. Docker, Shifter and Singularity are some

examples of Linux containers [2,9,10]. We can develop, deploy, test and run a scientific application in different computational environments with these containers. Containers have helped to improve numerical reproducibility by making it easy to copy the software environment of scientific applications. However, there are software patterns or idioms that produce inconsistent results because mathematical operations are performed in different orders in different settings. For example, some numerical computations in parallel distributed systems are not reproducible because math operations can be performed in different orders than on sequential processing systems.

A basic idea of science is that research results are reproducible. A hypothesis for future research is based on past results that are believed to be true. If the past results are not reproducible, scientists might develop new hypotheses based on false conclusions. This can lead to slower rates of discovery and inefficient use of resources, and it affects anybody who benefits from science, literally everybody. The rest of this paper is organized as follows. Section II provides background on some Linux containers and current work using algorithms to make application software environments reproducible. Section III is implementations and results from this research to improve reproducibility. Performance comparisons follows in section IV and conclusion and future research is in section V.

II. BACKGROUND & RELATED WORK

One of the main reasons that scientific application research is not numerically reproducible is that different computational platforms and architectures calculate mathematical operations in different ways. These differences are more significant when operations are performed inparallel distributed systems. For example, in heterogeneous computing different types of CPUs implement binary operations differently, which may affect the way numbers are rounded-off. Another possibility is that the type and/or version of the compiler that is used to compile the application may optimize (reorder) the code producing different sequences of instructions.

Experiments conducted by the National Center for Atmospheric Research (NCAR) show that containerizing Weather Research Forecasting (WRF) leads to identical outputs no matter what the computing system is. So far, NCAR has only reported results using single shared-memory nodes [13].

In the following sections, we discuss containers and reproducibility in scientific computing and the use of containers to improve reproducibility in parallel distributed systems.

A. Containers in Scientific Computing

The use of software containers has significantly grown in recent years due to their flexibility, scalability and portability. Linux containers have many of the characteristics that are supported by virtual machines (VMs), including isolation and customization. The main difference between a container and a VM is that containers are built directly upon the kernel, and for that reason they have a good level of isolation for memory and processors. However, this has some disadvantages. For example, the processes that are executing in containers will run

simultaneously in the same Linux kernel. This may affect the performance of the other applications or containers that are running on the same system.

Docker is one such technology that has emerged to manage containers with an image management system and are popular in the scientific community for many reasons including: simplified packaging of applications, promoted transparency of how the application is built, improved collaboration among researchers and supported reproducibility by insuring a consistent operation environment. However, there are some barriers to using Docker in HPC systems such as security issues, lack of compatibility with parallel file systems, limited scalability, and lack of integration with batch systems [4]. Finally, a performance study by IBM shows that Docker containers have a lower overhead compared to VMs, and this will significantly affect performance. According to the IBM study, "Docker equals or exceeds KVM [Kernel-based Virtual Machine] performance" in every single test that they conducted [7].

Containers promote reproducibility by providing a more consistent operating environment and make scientific applications portable to other researches [2]. Scientists may want to verify the results with another test, repeat a test sometime later, or they might want to duplicate their analysis with new data [4]. Containers help to improve numerical reproducibility in scientific research because they can be cited in published papers if that specific version of container exists and is available for the researchers to use in their modeling, analysis or implementations. However, there are still several factors affecting numerical reproducibility that containers cannot mitigate. For example, in truncation if the precision of data is up to 5 decimal points in one environment and up to 8 decimal points in another environment, there is a chance of different floating-point results. Baker et al. [6] show that there is an issue with the compiler optimization as the level of automatic optimization performed by a compiler can change the order of some operations.

B. Reproducibility in Scientific Computing

Recently, numerical reproducibility has received increased emphasis in the scientific community. For some scientists, numerical reproducibility means that the scientists can reproduce the same simulations every time that they run the application with the same conditions and configuration. This holds true even if scientists use different computer systems with different capabilities. For others, numerical reproducibility means that the results obtained when running on p processors p >= 2 should be the same as when running on a single processor.

Researchers want their application results to be reproducible [8]. However, when using parallel and distributed systems, program statements can run in slightly different orders on different processes. Also, when an application is moved to a different platform, reproducibility can be affected due to different operating systems, hardware, software, and network. In computing there is a certain amount of inexactness because there is a limit to how much memory can be used to store numbers.

There are several factors that can lead to reproducibility errors. Two factors, re-association and parallelism, have been studied in the literature.

1) Re-association and Compiler Options

The associative property in mathematics states (a+b)+c = a+(b+c). It means that we can associate a, b, and c using addition in any order that we want. Associativity also holds for multiplication. This is not true necessarily in finite precision computer programs.

In computing the order of operations matters because of finite precisions and limited amounts of storage for floatingpoint numbers. When we add or multiply two numbers, the result of one operation may be rounded or truncated before being used as part of the next calculations [3].

Two examples of reproducibility error due to reassociation were discussed by Corden and Kreitzer [3]. The first example comes from WRF and can be seen in Listing 1.

Listing 1. A loop in WRF

```
integer     i, n
real        B, TOL
real        A(1000), X(i)
Parameter (n-1000)
…
do 10       i=n
            X(i) = A(i) + B + TOL
10          continue
…
stop
end
```

In this code TOL is a very small, positive number. The programmer's intent for TOL is to keep X(i) positive if A(i) ≈ -B. An optimizing compiler can reorder the expressionshown in Eq. 1 for performance. The compiler may see (B + TOL) as a constant expression and move the expression outside the loop to reduce the number of floating point operations in the loop by one.

$$A(i) + (B + TOL) \quad (1)$$

Rounding and truncating the result of B+TOL effectively reduced TOL to zero, and X(i) was not always positive. When the compiler's safe mode was used, re-association was disabled and the compiler did not move B+TOL outside the loop.

A second instance of re-association affecting numerical reproducibility was also discussed by Corden and Kreitzer in an operation on arrays called reduction [3]. The code in Listing 2 shows a sequential sum reduction of an array.

Listing 2. Sequential sum reduction

```
int n=8;
float sum=0.0;
for (int i=0; i<n; i++){
      sum = sum + A[i];
}
```

With automatic parallelism using two compute units, the work of the compiler might be visualized as in Listing 3. The compiler unrolls the sequential loop to give work to each compute

unit. The statements in the body of the loop for i=0 are given to one compute unit, and the statements for i=4 are given to another compute unit. The value of each compute unit's *sum* is combined in shared memory. The parallel sum reduction adds array elements in different order than the sequential reduction. This can result in a different sum depending on the number of compute units used.

Listing 3. Parallel sum reduction using two compute units

```
int n = 8; float sum=0.0; for (int i=0; i<n; i=i+4) {
      sum = sum + A[0];
      sum = sum + A[i + 1];
      sum = sum + A[I + 2];
      sum = sum + A[I + 3];
}
```

Corden and Kreitzer used the elimination of parallelism to mitigate this type of reproducibility error. The suggested solution in their paper is to invoke a compiler option that effectively makes the compiler not to generate parallel code for reductions and only compute them sequentially. Generally, the result of sequential processing is considered correct when sequential and parallel processing have different results.

2) Recovery of Reproducibility Error

Numerical reproducibility is commonly determined by comparing results of a sequential computation with corresponding parallel computations. Langlois et. al, [12] consider three different techniques for recovering reproducibility to a summation type reduction. These techniques were integrated into a simulation program and reproducibility was measured using different numbers of processors. *Compensated summation* calculates rounding error generated during successive floating-point additions and adds the accumulated rounding errors to the total. The *reproducible sum* technique involves a pre-rounding step and parallel K-fold process with K chosen in advance. The final technique consists of converting every floating-point number to an 8-byte integer, adding the integers and reverting the sum back to a floating-point value. All these techniques return the same simulation results up to the computing precision for two to sixteen processors.

Langlois et. al, have also identified two broad categories of techniques for recovering reproducibility. In one category, the technique recovers reproducibility without improving accuracy. An example of this is using compiler options to disable the parallel reduction and just do the reduction with sequential processing. In this case, we avoid some reproducibility errors that are caused by parallel distributed processing, but we don't gain any accuracy in this way. The second category uses accuracy improvement to correct the rounding errors. In this case, the authors suggest a technique called the *accumulative of sums* approach. In this technique as we are adding a series of numbers, we are also calculating the reproducibility errors. At the end we take the total reproducibility errors and add it back to what the sum is. Therefore, the final results should be a very accurate number [11].

C. Containers and Reproducibility in Parallel Distributed Systems

There is a need in science for scalable and reproducible computing to be interoperated on different environments and platforms especially on parallel distributed systems or large scale HPC systems. One of the tenets of science is that results of experiments are reproducible. In scientific

computing, the software environment used to collect or analyze data must be reproducible to confirm that the results are reproducible [10].

One of these environments that provides HPC with containers is Shifter. Shifter has been derived from Docker at National Energy Research Scientific Computing (NERSC). NERSC is promoting the use of container technology for scientists on their systems [9]. Singularity is another environment that provides HPC for containers and has been developed by scientists at Lawrence Berkeley National Labs. Singularity is a container system that is portable and reproducible and has been designed for use on parallel and distributed HPC systems [10].

III. IMPLEMENTATION OF APPROACHES AND RESULTS

Software containers have been able to improve numerical reproducibility to some extent. However, we believe that we can improve the reproducibility of results of scientific applications within software containers even further. The hypothesis of this research is that applying some algorithms can improve numerical reproducibility of scientific applications within a Linux container by addressing rounding and truncation errors in certain mathematical operation in a reproducible manner. The detail of this work is provided is provided below.

A. Existing approaches to address reproducibility factors

One approach to calculate a reproducible floating-point summation in this research is particularly interesting. The main idea is to condition or pre-round the floating-point input data by removing less significant bits before summing the more significant bits. This technique sacrifices accuracy for performance. Pre-rounding involves extracting the high order parts of a value before accumulating the high order part in the sum. Adding a sufficiently large pre-rounding term to a value can isolate the high order part of the value. The pre-rounding term is then subtracted from that sum leaving the high order value [5]. For example, given a pre-rounding term M=1.030792e+11, and a value v=1.0e+10, the high order part, 9.999999e+09, can be isolated as shown in Eq. 2.

$$ho = (M + v) - M \tag{2}$$

The pre-rounding term does not need to be the same for each element of an array being summed. However, all prerounding terms must have the same unit of least precision, and the same directed rounding mode must be used. The high order parts and the sum of the high order parts will be reproducible [5]. Accuracy of the reproducible sum can be improved if the pre-rounding phase is repeated one or more times on the array of low order parts.

B. Develop implementations and combine them in a library

The following C++ implementation of an array summation has been developed as part of this study. The algorithm was previously published by Demmel and Nguyen [5]. Minor changes were made to reduce the number of arrays. Listing 4 shows code that sums the values from array v in a reproducible way. The sum is stored at the location pointed to by T, and array r contains the low order parts from array v.

Listing 4. Pre-rounding sequential reproducible summation of an array

```
void extract(float M, const std::vector &v,
                    float &T, std::vector &r) {
        float Mcurr = 0.0f;
        float Mprev = M;
        for (int i = 0; i < v.size(); i++) {
                Mcurr = Mprev + v[i];
                float ho = Mcurr - Mprev; // ho = high order part
                r[i] = v[i] - ho;                    // r[i] = low order part
                Mprev = Mcurr;
        }
        T = Mcurr - M;
}
```

The following code calculates the pre-rounding value, Mv[] used in function extract(). The sum, stored at location T, is made more accurate by calling extract multiple times. The original algorithm was also previously published by Demmel and Nguyen [5]. Listing 5 is an example of a sequential reproducible summation of an array that has been done so far for this research.

Consider a pre-rounding term M=1.030792e+11 and values a=1e+10, b=-1e+10, and c=1e-08. Pre-rounding a, b, and c results in 9.999999e+09, -9.999999e+09, and 0, respectively. The sum of the pre-rounded terms is 0. However, the exact sum a+b+c is 1e-08. The above values of a, b, and c can be also represented as: a = 10^{ϵ}, b = -10^{ϵ}, and c = $10^{\widehat{\circ}\epsilon}$. The summation orders of these values can be seen in Eq. 3. If a, b, and c are associated differently the exact sum changes. In Eq 4, the exact sum is now 0. The sum of the prerounded terms is still 0.

$$((a+b) + c) = ((10^{\epsilon} - 10^{\epsilon}) + 10^{\widehat{\circ}\epsilon}) = 10^{\widehat{\circ}\epsilon} \quad (3)$$

$$(a + (b+c)) = (10^{\epsilon} + (-10^{\epsilon} + 10^{\widehat{\circ}\epsilon})) = 0 \quad (4)$$

Listing 5. Pre-rounding sequential reproducible summation

```
float vMax = *std::max_element(v.begin(), v.end()); float deltaDenominator = 1.0f – 4.0f * (arraySize + 1) * FLT_EPSILON; float delta = arraySize * vMax / deltaDenominator; std::vector prTerms; prTerms.push_back( 3.0f *pow(2.0f, ceil( log2(delta))) ); std::vector sums; for (int i = 0; i < k; i++) { float pt = prTerms[ prTerms.size() – 1 ]; float eSum; extract( pt, v, eSum, v ); sums.push_back(eSum); Tv.push_back(*T); delta = arraySize * (4.0f * FLT_EPSILON * pt / 3.0f) / deltaDenominator; prTerms.push_back( delta ); } float mSum = prTerms[ prTerms.size() – 1 ]; for (int i = 0; i < arraySize; i++) { mSum = mSum + v[i]; } float lastSum = mSum = prTerms[ prTerms.size() – 1]; sums.push_back( lastSum ); float sum = 0.0f; for (int i = 0; i <= k - 1; i++) { sum = sum + sums[i]; }
```

We might consider summation of arrays across some processors. For example, we assume an array of twenty elements and four processors as shown in Figure 1. If the work is divided evenly, each processor will get five elements to sum. After calculating a local result, all processors will send their partial sum to one processor, and that processor will calculate the final sum. If the same array is divided among three processors as shown in Figure 2, the processors need to add a different number of elements. If the sum reduction were reproducible, then the same result would be obtained regardless of the number of processors used.

Figure 1. Twenty elements and four processors

Figure 2. Twenty elements and three processors

IV. PERFORMANCE COMPARISONS

Two runtime performance comparisons were made. The performance of the WRF application was measured in and out of a software container. Also, the performance of a reproducible sum reduction was measured and compared with a non-reproducible sum reduction.

A. WRF-Container Performance

In order to understand the overhead of Docker and how it might affect using Docker containers on a HPC system, the run-time performance of WRF was measured on both a Linux based system and as a containerized application. The results obtained show that using Docker is promising. However, overall performance improvements may be available when using different methods to access the application data.

The system used to measure performance was a 1.8 GHz Intel Core i5 processor with 8 GB RAM running Mac OS 10.13.4. Docker version 1.12.1 was installed for testing the containerized application. The WRF version 3.7.1 was used in both environments. The containerized WRF was obtained from a GitHub repository [13]. The data used in these WRF experiments was for demonstration and educational purposes and was smaller than the data that is used for weather prediction. The native version of the WRF binaries were created using instructions found online [1]. In the native tests and the containerized tests only one processor core was used.

Run-time performance data was obtained using the Unix time command. This command determines the duration of execution for a given command, in this case the WRF binary.

```
$ time ./wrf.exe
```

This command was entered at the system prompt for the native tests. For the container tests, the container was modified to run the time command with the WRF binary as shown above. The times for five runs were measured in each environment.

The Unix time command reports three different time results: real time, user time, and system time.

- The real time or wall-clock time is the total time used by the application from start to finish. This may include time while the program is not executing, but is waiting for another application.
- The user time is the total CPU time spent running the program. This does not include times for system calls such as I/O.
- The system time is the time spent performing system calls on behalf of the application. We can think of system time as kernel time.

Table 1. Results of Individual runs of WRF application on the Linux OS and in a Docker container. All times are in seconds.

	Native User	Native Sys	Docker User	Docker Sys
1	186.681	1.840	189.466	13.775
2	187.287	1.895	190.167	13.745
3	187.591	1.759	191.239	13.717
4	186.775	1.821	189.07	13.440
5	186.923	1.863	189.718	13.519
Avg	187.0514	1.8356	189.932	13.6392

The run-time performance of sets of individual runs and the average times of each set can be seen in Table 1. Native User and Native Sys columns show the user and system times of the application running on the Mac operating system. The Docker User and Docker Sys columns show the user and system times, respectively, of the containerized WRF application on the same Mac operating system. The results are in seconds.

A chart showing the average times in each category is shown in Figure 3. The y-axis is in logarithmic scale. The results obtained show that the native user time is approximately 1% faster than the containerized application. This shows that Docker does not contribute a significant overhead to executing programs.

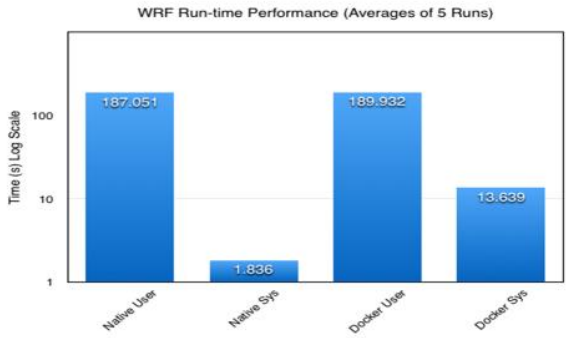

Figure 3. Comparison of Native and Container Application Performance

The system time between the native and Docker environments is significantly different. The Docker system time is 7.4x slower than the native system time. This difference may be caused by differences in how the native and containerized WRF applications accessed the WRF data. In the native version the data was stored on the local file system. In the Docker version the data was in two separate data containers. Further tests would be needed in order to understand more completely how data access affects run-time performance and what steps can be taken to minimize data access times.

B. Reproducible vs Non-reproducible Sum Performance

The performance of a non-reproducible sequential sum reduction implementation was compared with a fast sequential K-fold reproducible sum reduction [5] in standalone programs. The programs were compiled with GCC compiler version 6.1.0 using the compiler –O1 option. This option disabled any automatic vector processing that may have occurred. Both test

programs were run on a single core of an Intel Xeon E5-2643 (3.3GHz) with 256 GB RAM. The system ran RHEL 7.2.

The K-fold value for the reproducible sum reduction indicates the number of times the floating-point values in the array are pre-rounded. On the first time the original floatingpoint values are pre-rounded. On subsequent times, the less significant bits are pre-rounded. This can increase the accuracy of the sum reduction. However, the K-fold value was always one in these performance tests in order to be similar with the non-reproducible sum reduction.

The reproducible sum reduction algorithm requires a directed rounding mode. Therefore, the rounding mode for each program was programmatically set to directed *rounding to zero* using the fesetround function (cenv library).

The array size was specified with a command line option, and an array of floating-point values to sum was initialized. Elapsed time was computed using the difference of gettimeofday function calls (see the sys/time.h library) before and after the sum reduction code. This did not include the time for generating the array of floating-point values to be summed. The programs measured the sum reduction performance five times.

The performance of each implementation was measured with array of 1,000 to 1,000,000 elements. The reproducible sum reduction was 0.30x to 0.47x the runtime speed of the non-reproducible sum reduction as shown in Figure 4. This is consistent with the number of floating-point operations in the non-reproducible sum reduction, O(n), compared to the floating-point cost of the reproducible sum reduction, (3k - 1)n + O(1) [5].

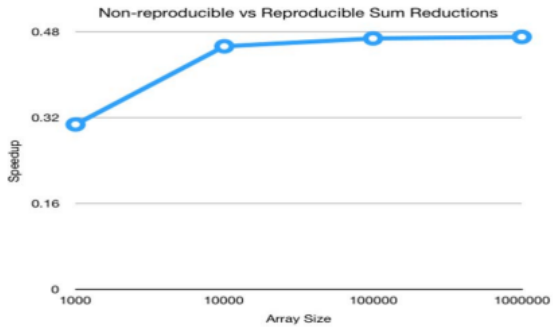

Figure 4. Reproducible sum reduction is over 2x slower than the non-reproducible version.

V. CONCLUSION AND FUTURE WORK

The use of software containers is growing rapidly across multiple domains of computing today. Containers provide a consistent, lightweight operating environment for applications that can be easily shared with colleagues which helps make results reproducible. However, copying containers does not address mathematical operations that are not reproducible due to truncation and rounding of floating-point results.

This research has helped to understand the performance impact of two reproducibility mitigation techniques. Running WRF inside a Docker container was found to perform 0.98x the runtime performance of running WRF in a native Linux environment. Containerized applications accessing data in other containers can expect reduced performance. An implementation of an algorithm for a reproducible floatingpoint sum reduction was between 0.30x and 0.47x the runtime performance of a non-reproducible sum reduction. The reduced performance was due to calculating a pre-rounding term, applying that term to every array element, and recovering the low order parts each array element not included in the sum reduction.

This research improves numerical reproducibility further by finding mathematical operations in source codes that are reproducibly problematic. These operations can be replaced by calls to the operations that are reproducible. This improvement extends the reproducibility gained when run within software containers. As a result of this research, numerical reproducibility of scientific applications will be improved. This can lead to faster rates of discovery and efficient application of scientists' time and grant money.

Our intent is to continue this research by developing sequential and parallel implementations for each reproducibility approach and combine them in a library that can be included in a container. Then developing a source code scanner to recognize numerical reproducibility factors and replace those factors with calls to the corresponding reproducible implementation in the previously mentioned library. Also, after implementing a parallel summation, similar sequential and parallel operations will be developed for subtraction, multiplication, division, square root, fused multiply-add, and the remainder operations.

REFERENCES

[1] University Corporation for Atmospheric Research, "How to Compile WRF: The Complete Process" http://www2.mmm.ucar.edu/wrf/OnLineTutorial/compilation_tutori al.php

[2] C. Boettiger, "An introduction to docker for reproducible research," ACM SIGOPS Operating Systems Review, vol. 49, no. 1, pp. 71–79, 2015.

[3] Corden M J and Kreitzer D "Consistency of floating-point results using the intel compiler or why doesn't my application always give the same answer" (Technical report, Intel Corporation, Software Solutions Group). 2009

[4] R. S. Canon and D. Jacobson "Shifter: Containers for HPC". NERSC, Lawrence Berkeley National Laboratory, 2016.

[5] J. W. Demmel and H. D. Nguyen, "Fast reproducible floatingpoint summation," in Proc. 21th IEEE Symposium on Computer Arithmetic. Austin, Texas, USA, 2013.

[6] A. H. Baker, D. M. Hammerling, M. N. Levy, H. Xu, J. M. Dennis, B. E. Eaton, J. Edwards, C. Hannay, S. A. Mickelson, R. B. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson, 2015: A new ensemble-based consistency test for the Community 439 Earth System Model (pyCECTv1.0). Geosci. Model Devel., 8, 2829–2840, DOI:10.5194/ 440 gmd-8-2829-2015.

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An Updated Performance Comparison of Virtual Machines and Linux Containers,” Technology, vol. 25482, 2014.

[8] W. Gropp and E. L. Lusk. Reproducible measurements of MPI performance characteristics. In Proceedings of the 6th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 11–18, London, UK, UK, 1999. Springer-Verlag.

[9] D. M. Jacobsen and R. S. Canon, “Contain this, unleashing docker for hpc,” Proceedings of the Cray User Group, 2015.

[10] Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PLOS ONE. 2017;12(5):e0177459 doi: 10.1371/journal.pone.0177459

[11] P. Langlois, R. Nheili, and C. Denis, "Recovering Numerical Reproducibility in Hydrodynamic Simulations", IEEE 23nd Symposium on Computer Arithmeticpp. 63-70, 2016, doi:10.1109/ARITH.2016.27

[12] P. Langlois, R. Nheili, and C. Denis, “Numerical Reproducibility: Feasibility Issues,” in NTMS’2015: 7th IFIP International Conference on New Technologies, Mobility and Security, Paris, France, Jul. 2015, pp. 1–5.

[13] John Exby, “Docker-WRF” https://github.com/NCAR/container-wrf, retrieved on Sept. 12, 2016