A CONTAINERIZED MESOSCALE MODEL AND ANALYSIS TOOLKIT TO ACCELERATE CLASSROOM LEARNING, COLLABORATIVE RESEARCH, AND UNCERTAINTY QUANTIFICATION

Joshua P. Hacker, John Exby, David Gill, Ivo Jimenez, Carlos Maltzahn, Timothy See, Gretchen Mullendore, and Kathryn Fossell

Software containers can revolutionize research and education with numerical weather prediction models by easing use and guaranteeing reproducibility.

Numerical models are a cornerstone of weather prediction today, and they support a broad range of weather research. Studies quantifying the ability of numerical models to predict the atmospheric state, and to simulate atmospheric phenomena, form two key lines of inquiry. By establishing model fidelity to the atmosphere, those studies also provide a basis for broader scientific inquiry with models. The use of numerical models in atmospheric research has become ubiquitous over the past few decades. Areas of research include physical process identification and analysis, atmospheric predictability, and predictions of future climates, among others.

Extensive use of numerical models in research demands educating students for diligent application of these large and complex codes. Beyond basic theory and best practices in numerical methods, the opportunities for mistakes are vast. While mistakes can provide positive learning experiences, they can also produce misleading results and consequently incorrect interpretation.

The purpose of this article is to describe the implementation of software containers for numerical weather prediction (NWP) research and education. Container technology has profound implications for education and research in numerical weather prediction. Containers not only enable reproducibility, but they greatly lower barriers for accessing cutting-edge NWP codes. The tools discussed herein, such as source code repositories and containers, may be relatively new for many readers. These tools are presented in detail to fully describe the procedures used. But a deep understanding is not needed. The containers presented here have already been tested successfully by multiple users, primarily from universities. The technical details can remain mostly transparent for new users; the power of these tools can then be further realized by users (including students and educators) who become more advanced.

The Weather Research and Forecasting (WRF) Model (Skamarock et al. 2008) is a state-of-the-art NWP model for operations and research, with users numbering in the tens of thousands. It was engineered to be portable and easy to install on a limited yet common set of platforms, with code that is both complex and extensive (over 1.5 million lines). It has many dependencies on external software packages [e.g., for input/output (I/O), parallel communications, and data compression] that are not trivial to satisfy; compilation and execution can be an intensive effort for beginning users who lack support from experienced system administrators. The WRF help desk requests are dominated by new users reporting difficulty compiling the model. A new version has been released twice annually for the last 15 years, and users who want to update and rebuild may face repeated challenges with each new set of code.

Running the model can also be difficult for new users. Individual steps to generate computational grids, import initialization data, produce initial and boundary conditions, and run the model can be daunting. Those steps can be scripted into a workflow that can range from simple to complex, depending on the application. Scripted workflows have been developed by countless individuals and groups over the last decade; the proverbial "wheel" has been reinvented countless times, especially when considering the number of individuals who have written analysis tools to compute the same diagnostics on output.

Implementation of experiential learning with numerical weather prediction codes is especially challenging for users in the classroom. Some universities are able to provide hands-on exercises due to a combination of sufficient laboratory resources and, most importantly, staff (faculty and support) with a sufficient knowledge base in modeling and information technology. But even those universities with sufficient knowledge struggle with continual updates to compilers, operating systems, and model code.

Results from software that implements floating-point operations are generally not numerically reproducible. Examples of the impacts on numerical weather models can be found in Thomas et al. (2002) and Baker et al. (2015). A consequence is that NWP research is not reproducible.

Numerical uncertainty can lead to misleading results. The chaotic nature of the models means that small perturbations, perhaps arising from truncation errors, can organize and grow into features that are tempting to interpret as an important response to a change in input or model formulation. The classic paper by Lorenz (1963) is an example of how numerical truncation can lead to chaotic results, and in that case it was truncation that led to Lorenz's discovery of chaos. Many times the differences that appear meaningful cannot be reproduced, or they simply cannot be distinguished from chance.

Finally, collaboration among researchers working on different computing platforms can be cumbersome. It often relies on careful manual code management to make sure collaborators are working with the same code. Configuration of a complex model must also be done carefully—and still the opportunity to make mistakes is always present.

Software containers, which are becoming an essential part of modern software development and deployment, offer a path for mitigating or eliminating many of the problems in NWP research and education that are described above. In the remaining text, we describe how and show some examples. First, software containers are reviewed. Then a set of containers for initializing, running, and analyzing results from the WRF Model are presented. Numerical and performance reproducibility are demonstrated with a simulation of Hurricane Sandy, leading to a clear way to perform uncertainty quantification in the following section. The discussion section concludes with some analysis of the implications for container technology in numerical simulation and prediction. In the final section, links to online content and open repositories are provided.

SOFTWARE CONTAINERS. A container is a software-based packaging and distribution tool that collects all elements and dependencies of a Linux-based application. Containers store the runtime environment together with one or more applications for ease of transportation, installation, and execution across a variety of operating systems (Linux, Mac, Windows). Containers avoid any need

AFFILIATIONS: Hacker, Exby, and Gill—Research Applications Laboratory, National Center for Atmospheric Research, Boulder, Colorado; Jimenez and Maltzahn—Computer Science Department, University of California, Santa Cruz, Santa Cruz, California; See and Mullendore—Department of Atmospheric Sciences, University of North Dakota, Grand Forks, North Dakota; Fossell—Mesoscale and Microscale Meteorology Laboratory, National Center for Atmospheric Research, Boulder, Colorado
CORRESPONDING AUTHOR: Joshua P. Hacker, hacker@ucar.edu

The abstract for this article can be found in this issue, following the table of contents.
DOI:10.1175/BAMS-D-15-00255.1

In final form 24 August 2016

for recompilation or complex virtual machine environments. Because they house only the necessary components to execute an application, they are much simpler and smaller than virtual machines running complete operating systems. Containers allow an application (or suite of applications) to be developed, tested, and executed in such a way that when relocated to an alternative compute environment, the execution and results of the application workflow are consistent and repeatable. Modern container tools have origins based on Linux containers that have long been part of Linux kernels. They are now evolving toward a standards-based "engine" that allows portability and compatibility despite changes to underlying compute systems, networks, or operating systems.

An image, or base image, is a prebuilt set of application elements—binaries, libraries, configuration files, and any ancillary tools—required to provide an entire execution environment for the application to function. An image can be developed over time by being tagged and maintained with version control methods, and downloaded from official repositories as a basis for developer or operational use. An application image may be maintained by a known set of developers for the purposes of official verification at known version levels, and contain official patches and security updates.

Launching a particular instance of an application image into a run-time state produces a container running that application. The container is executing the entire environment of the application, without requiring the host operating system to have any software installed other than the container engine itself.

Linux containers have been used in cloud services for over 10 years to provide stability, scalability, and a more secure, isolated run-time environment than is often possible on commonly used computing platforms. Today's container engines have defined an emerging standard that can execute prebuilt codes within every modern operating system (OS), including Windows, Mac OS X, and Linux. Containerized applications provide a well-documented command set that users can implement quickly without regard to a destination OS. Software deployment methods are simple and reduce the level of introduction needed to bring new users or developers into a collaborative group.

Industry experts believe containers will continue to be prevalent, with accelerating adoption due to best-fit scaling on desktops and laptops. Commercial and research cloud platforms are providing greater off-the-shelf server resources as hardware prices either fall or computing power per dollar spent rises. Container technology is a natural fit to commercial cloud services; a user can immediately deploy software without any knowledge of the underlying hardware or OS. Once software tools are in containers, users have easy access to vast resources of cloud computing, which can be exploited when conditions demand.

We chose Docker (www.docker.com) as the container engine for this work, but others are available. Docker containers are user friendly, already widely adopted across the software industry, and allow users to execute complex precompiled scientific codes anywhere. The combination of all these features enables users to focus on running numerically reproducible scientific analysis and to reduce start-up times. Within containers, software configurations can be easily integrated into arbitrary workflows that assemble trivially portable modules. The containers may use differing datasets assigned at launch and may be built on demand. The resulting output can be stored with metadata using local storage; proven distributed storage technologies; cloud storage, such as Amazon Simple Storage Service (Amazon S3) and Amazon Glacier offered by Amazon Web Services (https://aws.amazon.com/); or similar resources.

The open-source community and the rapid adoption of Docker (and similar) container technologies have boomed since 2012, with entire companies, such as Uber, Netflix, Goldman Sachs, and others, leveraging containers across hybrid (on premise and cloud) platforms for dynamic scalability that can be based on users (or events) worldwide. Frameworks for managing complex and multiple instances of containers are evolving to provide streamlined tools that allow flexibility, verification, reporting, and coordination.

Containers internally run a very lightweight Linux environment, as opposed to a full Linux kernal supporting an operating system. Prior to June 2016, the stable released Docker engine software components for Windows and Mac OS X required an additional thin virtual machine (VM) layer, managed by VirtualBox (www.virtualbox.org), to pass container commands to a Linux VM environment that supports containers. The VM layer added a slight performance loss (approximately 5% in run time), and it created extra Docker Machine configuration and management steps often not familiar to general users. As of June 2016, new (version 1.10 + beta and beyond) versions of the Docker engine for Windows and Mac OS X have bypassed the requirement of a Linux VM, completely discarding Virtualbox and all local Docker Machine management and VM configuration steps. Docker containers now run natively on Windows and Mac

OS X platforms, eliminating the slight performance penalty of VMs and delivering container commands to native Windows and Mac OS X kernels. Docker continues to build upon this new beta software for the purposes of incorporating the same native features across Linux, Windows, and Mac OS X.

A SET OF CONTAINERS FOR NUMERICAL WEATHER PREDICTION. The Docker containers described above have been implemented so that a user can easily configure and complete a WRF run. An example deployment of the containers is summarized in the schematic in Fig. 1. The Docker-WRF container also includes the WRF and the WRF Preprocessing System (WPS) to ensure software version consistency and compatibility. A separate container with the National Center for Atmospheric Research (NCAR) Command Language (NCL; www.ncl.ucar.edu) plotting scripts is also implemented. Easy run-time configuration is provided by external access to the WPS and WRF namelists on the local file system. Output and namelists (to record provenance) are written to the local file system, and additional metadata can be easily included. Prebuilt data containers (including WPS and WRF namelists) for specific simulation periods and regions are currently available in open repositories. In that case the local folders for the namelist, input files, and static geographic (GEOG) data are unnecessary. Containers for acquiring input data, archiving output, and sharing output are under development in collaboration with several universities under the National Science Foundation (NSF)-funded Big Weather Web project (http://bigweatherweb.org).

The WPS–WRF container currently holds the WPS and WRF code and the prebuilt binary executables (applications). The container includes executables to create domains, populate files with geographical and land surface data, interpolate input meteorological data to the WRF computational grid, and run the WRF to produce a simulation or forecast. GNU compilers and all dependencies are included. A user launches the container with command line options for recompiling the code and executing the various WPS and WRF executables. It is simple for users who want to make code modifications to enter the container and make changes before recompiling, or to recompile within the container before exiting and executing WRF runs from the outside.

Figure 1 is an example of one possible workflow. It is easily scripted or run from the command line. The intent is for users to be able to deploy any number of the containers and to substitute their own at any step in the workflow. For example we can imagine that a user may want to introduce a different and more sophisticated postprocessing approach that allows for analysis supporting a physical process study. That new container would be deployed in exactly the same way as long as it only needs access to the WRF output files. The results would be archived or shared in the same way, and the new container can be uploaded to the repository for community use and reproducibility.

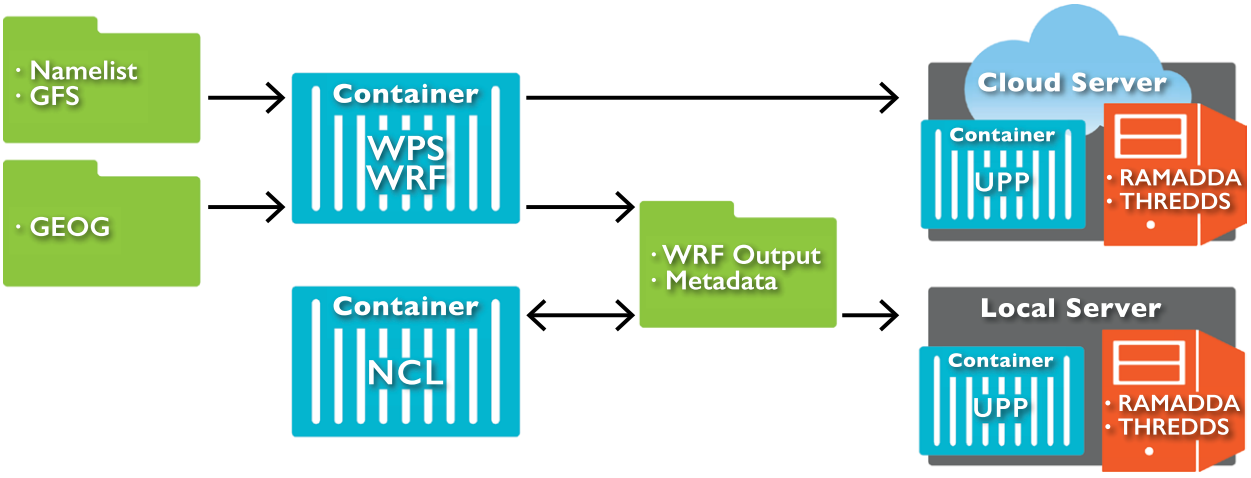

Fig. 1. Schematic illustrating an example deployment of the WPS and WRF in a Docker container. Green folders denote files on a local file system that are easily made available to the containers or a local user. This example uses GFS output files for WRF initial conditions and lateral boundary conditions. The containers (blue) interact with the local file system along the paths shown with arrows. Output and metadata can be stored and served with local resources, or in a hosted computing and storage environment (the cloud). Data can be served in either case with help from, e.g., the Geode Systems's Repository for Archiving and Managing Diverse Data (RAMADDA) or Unidata's Thematic Real-Time Environmental Distributed Data Services (THREDDS) software.

In the spirit of a community and open-source effort, users can contribute useful containers so that others can reproduce results or address other scientific questions with similar tools. The question a researcher often faces, of developing his or her own code or trying to adapt and port code developed by others, will go away in many cases because no porting effort is required.

Source codes for the model and ancillary systems, the containers, and datasets (e.g., the geography data needed to design WRF domains) are managed in version-controlled source repositories, which allow continuous integration and unit testing through the chain of source commits to Docker-build image generation. We chose GitHub (www.github.com), but we could have equally chosen another repository host, such as Bitbucket (www.bitbucket.org) or GitLab (https://gitlab.com). The GitHub repositories are for developers, or for researchers who want to rebuild containers to address specific scientific questions. When a project on GitHub is executed to produce a Docker image, the resulting image is pushed to Docker Hub, which manages Git repositories for Docker images. We could have equally chosen an Amazon or Google registry service for hosting the Docker images. A user who is not interested in developing (e.g., changing the WRF code or analysis tools) can simply pull images and use them to run containers with any necessary run-time input files (namelists, gridded data, etc.). Keeping all known fixes and build environments updated, patched, and tested through a quality assurance process via a core development team, including code commits to a GitHub repository and pulls from a Docker Hub image repository, guarantees software integrity in the launch environment for model results to be replicated, or queried, for future studies.

Collaboration on research, which may or may not require code changes, is easily enabled via Github, Docker Hub, and the reproducibility properties inherent to working in containers. Collaborators can maintain a repository with version control on the code and container. Information and links on how to obtain the source for the containers, or the images, on open GitHub and Docker Hub repositories are given in the resources section at the end of this paper. The version control system ensures that bit-perfect results can be obtained across native OS platforms, and that users are accessing an identical code base as required. All collaborators in a group can be sure they are using identical codes and even binaries, providing unambiguous interpretation of quantitative analysis of output from the model or a set of analysis tools.

NUMERICAL AND PERFORMANCE REPRODUCIBILITY. One of the primary reasons that NWP research has historically been numerically nonreproducible is that different computer architectures and environments perform mathematical operations differently. Differences can be greater when parallel operations are included. First, different chips have different binary truncation, leading to different effects of round-off error. Second, compilers sometimes reorder operations in attempts to speed computation; the reordering depends on the compiler brand, version, and level of optimization. Third, unless carefully implemented, parallel operations may not be performed in the order intended. Often this is not a requirement, but truncation guarantees that some computations are not reproducible when the order of a parallel computation is not enforced. For example, the result of an average depends on the precision in the sum and division, and in which order they are performed. Thomas et al. (2002) demonstrate the effect of compiler optimization and parallel topology on NWP, and Baker et al. (2015) show how the lack of expected numerical uncertainty can indicate deficient code quality. Experiments described here test uncertainty introduced by different hardware and operating systems, parallel topology, and compiler options.

A computationally inexpensive WRF simulation of Hurricane Sandy is the basis for experimentation. The simulation is on a single computational 50×50 horizontal domain with $\Delta x = 40$ km and 60 vertical levels. Initial and boundary conditions are interpolated from National Centers for Environmental Prediction (NCEP)'s Global Forecast System (GFS). Twelve-hour simulations are initialized at 1200 UTC 27 October 2012. Details of the complete subgrid and forcing schemes (known as "physics") are unimportant here, but they do represent a reasonable set with components widely found in the literature. This computationally inexpensive simulation serves to illustrate the capability. A plot of total precipitation accumulated over the 12-h simulation, shown in Fig. 2, is also useful to see the computational domain. The very same container demonstrated here can be trivially used to execute other WRF simulations simply by changing the input data and namelists.

The Sandy container simulation has been tested on dozens of different computers with a variety of hardware and operating systems. Table 1 shows a small representative sample, including common Mac and Linux platforms. It also includes resources available on Amazon Elastic Compute Cloud (EC2) and Packet's (www.packet.net) Tiny Atom cloud

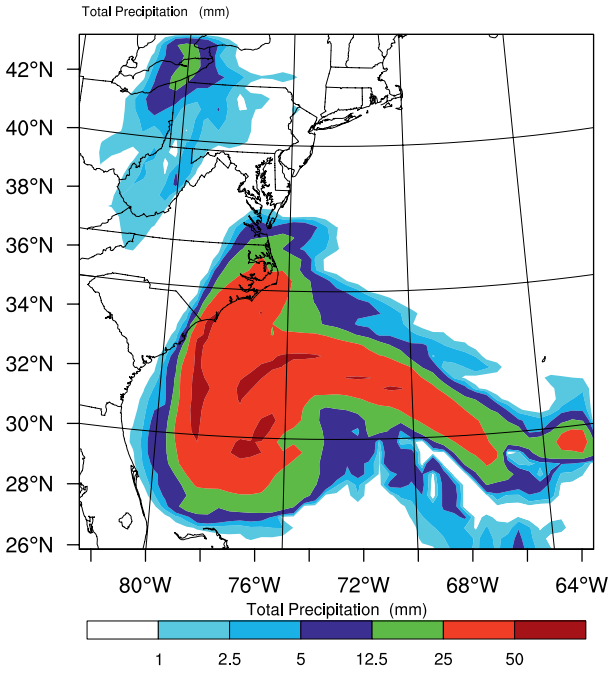

Fig. 2. Simulated 12-h total precipitation from the demonstration Hurricane Sandy simulation, valid 0000 UTC 28 Oct 2012.

computing infrastructure. Successful deployment and testing is far more extensive than shown in Table 1.

In each case, the same container is deployed and executed. Hourly WRF output files are saved on a user's file system by externally linking to the WRF output directory within the container's file system. An arbitrary platform is chosen as a reference, which serves as comparison for all of the other test deployments. A deployment in Mac OS X 10.10.5 (Yosemite), with an Intel Xeon E5 on a Mac Pro, serves as the reference here. Free GNU compilers—namely, GFortran and GNU Compiler Collection (GCC)—are part of the container for building the WRF executable. Because the compiler and compiler flags are the same, the binary WRF executable is identical regardless of what computer compiles it. But it is not necessarily obvious that the same binary will produce the same output on different hardware.

Numerical differences between the reference deployment and any other deployment are simple to compute with any variety of binary difference utility, such as the UNIX cmp or checksum, or the netCDF differencing operator ncdiff. In all testing so far and regardless of the computing system, all of the WRF output files from the same configuration are bitwise identical.

A container that builds the WRF internally, via the GNU compilers that are inside the container, can be equally deployed on any platform that supports containers. Experiments have shown that those results are also identical to the output from prebuilt binary executables deployed as above.

Container deployment on individual servers, regardless of the number of CPUs or computational cores, also yields identical results. Both the container technology and that the WRF code is sufficiently mature to make the same calculations on any parallel topology are responsible for that result. Numerical reproducibility across different parallel topologies is nontrivial for such a complex code (e.g., Thomas et al. 2002), and the WRF developers deserve credit. We have not tested all possible configurations of the WRF, which number in the tens of thousands, so we cannot guarantee that all configurations are robust to differences in parallel topologies.

Results from the cross-platform testing here show that numerical reproducibility is easy to achieve with containers, within some easily controlled limitations that are discussed next. By wrapping all dependencies in a container, researchers working on disparate local hardware, OS, and compiler resources can obtain numerically identical results.

Containers are an attractive upgrade from virtual machines because the computational overhead is significantly less. In an extensive analysis, Felter et al. (2014) found that configured properly, Docker containers surpassed kernel-based virtual machines (KVM) in every performance test. Computational and memory performance overhead is very small, and any performance penalty is on I/O and interactions with the operating system. Although OS interactions are negligible for NWP models, I/O can be a limitation. Fortunately the overhead is dependent on the number of I/O operations, not the I/O size. In many relevant problems, the WRF does not output many files, but it can output large files for large computational domains.

Table 1. Representative sample of container test platforms for the Hurricane Sandy simulations. AMD: Advanced Micro Devices Inc. RHEL = Red Hat Enterprise Linux.

OS	Chip/CPU	Cores
Mac OS X 10.10.5	Intel Xeon E5	6
Mac OS X 10.9.5	Intel Core i7	2
Ubuntu 14.04	AMD Opteron 6320	16
Ubuntu 14.04	Intel Xeon E5	16
RHEL	Intel Xeon X5550	8
Ubuntu 14.04[a]	Intel Xeon E5-2666, version 3	32
Ubuntu 14.04[b]	Intel Atom C2550	4

[a] Deployment on an Amazon EC2 resource; the others are on a laptop or desktop/server.

[b] Deployment on a Packet Tiny Atom resource.

In tests on the Sandy simulation we also find minimal performance impact from running within the Docker container. Identical binary executables were run four times on the same machine both inside the container and outside the container on a Linux server. Results showed an average 3.6% overhead on wall-clock time when running in the container. The overhead will be greater for operating systems requiring a VM layer (e.g., Windows) with older versions of Docker.

This Sandy simulation requires relatively few floating-point operations, compared to model executions in typical research and operational NWP today. It is parallelized on shared-memory computer servers. On a MacBook Air as described in the second line of Table 1, the sequence of pulling the container from the repository, running the WRF, and making plots takes approximately 4 min. The example demonstrates the potential for classroom utility.

Performance reproducibility can be contrasted with the numerical reproducibility examined above. Performance reproducibility deals with the issue of obtaining the same performance (run time, throughput, latency, etc.) across executions. Reproducing performance is important for gaining accurate evaluations of that performance. For example, when performance improvements to the WRF code base are implemented, evaluating improvements is difficult because it is difficult to account for the changes in hardware over time. In Jimenez et al. (2016), the authors introduce a framework to control the cross-platform variability arising from changes in CPU architectures and generations. This technique leverages Linux's cgroups technology (an underlying component of Docker), which allows a user to specify the CPU bandwidth associated with a container. For example, one can constrain a WRF container to have 50% the capacity of a CPU core (instead of 100% of the bandwidth available, the default behavior). With this feature, one can find the amount of CPU capacity that emulates the behavior of a slower (usually older) machine. Table 2 shows the results of applying this technique to reproduce the performance of the Hurricane Sandy simulation on multiple platforms.

In Table 2, machine issdm-6 is the base system where a reference simulation was executed. When this same simulation is reexecuted in other (newer) machines, an inherent speedup in run time results from the improvements in newer CPU technology. For example, machine dwill improves the run time of the original execution by approximately 2.8 times. Using the CPU calibration techniques introduced in Jimenez et al. (2016), which leverage the OS-level virtualization in containers, we carefully constrain the CPU of the machines and reexecute the simulation. As described in Jimenez et al. (2016), the limits on CPU are obtained by executing a battery of micro-benchmarks that characterize the CPU performance of a base system (issdm-6 in this case). Benchmark results are used to calibrate the target machine (for example dwill) by finding the percentage of CPU

Table 2. Run time of container executions for the Hurricane Sandy simulations on multiple machines. The normalized run time is the ratio between the reference system issdm-6 and the platform identified in the particular row. In the CPU limit column, 100% denotes no limitations imposed on the CPU, i.e., the simulation had all the CPU capacity from the system.

Machine identification	CPU model	Run time (s)	CPU limit (%)	Normalized
issdm-6	AMD Opteron 2212	1251.697012	100	1.000000
packet0	Intel Atom C2550	1173.848259	100	1.066319
packet0		1265.819255	99	0.988843
nibbler	Intel i7 930	571.163721	100	2.191485
nibbler		1425.671584	63	0.877970
dwill	Intel i5 2400	442.803647	100	2.826754
dwill		1244.367972	38	1.005890
rackform4	AMD Opteron 6320	689.738835	100	1.814741
rackform4		1560.987534	62	0.801862
pl2	Intel E5-2630 V2	425.031332	100	2.944952
pl2		1431.383836	48	0.874466
node	Intel E5-2630 V3	396.553413	100	3.156440
node		1564.413064	56	0.800106

capacity it needs to execute the same list of micro-benchmarks and emulate the performance of the base system. In Table 2, the CPU limits column denotes the tuned amount of CPU capacity that the machine was given for a simulation, with the goal of emulating the performance behavior of the base machine. For the case of dwill, the limitations on CPU bandwidth (38% of CPU capacity) bring the runtime down (introduce an artificial slowdown), closely resembling the original performance.

Performance reproducibility also enables planning for computing needs. This is becoming more important as cloud computing becomes more prevalent. Commercial cloud vendors such as Amazon Web Services, Packet, Google Kubernetes (https://kubernetes.io), and Microsoft Azure (https://azure.microsoft.com/en-us/) generally support container deployment. A researcher or company writing a proposal that may include a need for cloud computing resources can get accurate estimates of performance and expect it to hold onto the variety of hardware offered by vendors.

This section presented a subset of results from runs on many different machines. Although parallel WRF runs have been part of the testing, they are limited so far to single shared-memory nodes. Distributed (multiple nodes or servers) message passing interface (MPI) for WRF within Docker containers is under investigation. Progress depends on the MPI-based WRF application having awareness of other linked WRF executables in other containers that may be part of a collective container group. Currently MPI does not have that capability, although Docker containers can be made aware of parallel containers within a collective.

UNCERTAINTY QUANTIFICATION. A corollary to numerical reproducibility is that a precise quantification of uncertainty from other factors is enabled. The reproducibility means that by leveraging available computer power wherever it is available, hundreds or thousands of simulations can be produced with perturbations introduced in initial conditions or model configurations. Such an ensemble would reflect only the perturbations given to it and would not be contaminated by random or systematic errors introduced from traditional computing platforms or changing compiler options. Leaving ensemble investigations for future work, here we examine the effects of compiler optimization. The effects we report here are certain to be reproducible across many computing platforms.

A simple pair of simulations illustrates the effects of compiler optimization on numerical predictions. The basic Fortran optimization flag for the GNU compiler was successively set to levels from 0 to 3, which is a typical range for compilers. Setting -O0 forces the compiler to avoid optimizing calculations and memory access, and setting -O3 allows the compiler a number of relatively aggressive optimization strategies. Those may change the order of operations, for example. Different compilers introduce different optimizations at that level. We found in this specific case that levels 0–2 led to a WRF executable that produced identical results, but level 3 produces differences. One of the optimizations introduced at level 3 has an effect on output, but it is beyond the scope of this work to determine exactly what that is. Instead, we simply compare the output between levels 0 and 3.

A histogram of 3D gridpoint differences between the two Sandy simulations, after 12 h of simulation time, shows that meaningful differences are present (Fig. 3). Although the majority of differences are small, the distribution tails indicate the onset of local perturbations. We know those perturbations will grow and propagate upscale because the model equations are chaotic (e.g., Lorenz 1969). We can expect that higher-resolution grids, subject to more small-scale nonlinear processes, will lead to greater rates of growth of the differences between the two simulations. Grids covering larger regions will be subject to greater rates of growth too, because the dynamical interactions cover a broader range of scales.

DISCUSSION. This article presents a set of open-source Docker containers intended to provide a basis for a WRF execution and ancillary workflow components. As shown above, the container infrastructure offers many advantages for enabling research and education to be reproducible and more collaborative. A few more discussion points are summarized here.

The containers offer numerical and performance reproducibility in an easily—and rapidly—deployable framework that can enable collaborative research and education. In the earlier section on uncertainty quantification, we demonstrated the effects of compiler flags. Many NWP experiments reported in the literature focus on small differences between simulations, which are interpreted as physically relevant signals to be diagnosed. Although the majority of the differences may be physically relevant, the reproducibility provided by containers offers certainty. The extension is that uncertainty from sources besides numerical truncation can be isolated. To further understand these issues, a study evaluating whether compiler optimizations, hardware variations, and parallel topologies lead to random-like or systematic errors would be helpful.

The containers enable the same kind of collaborative environment as a single community resource, but

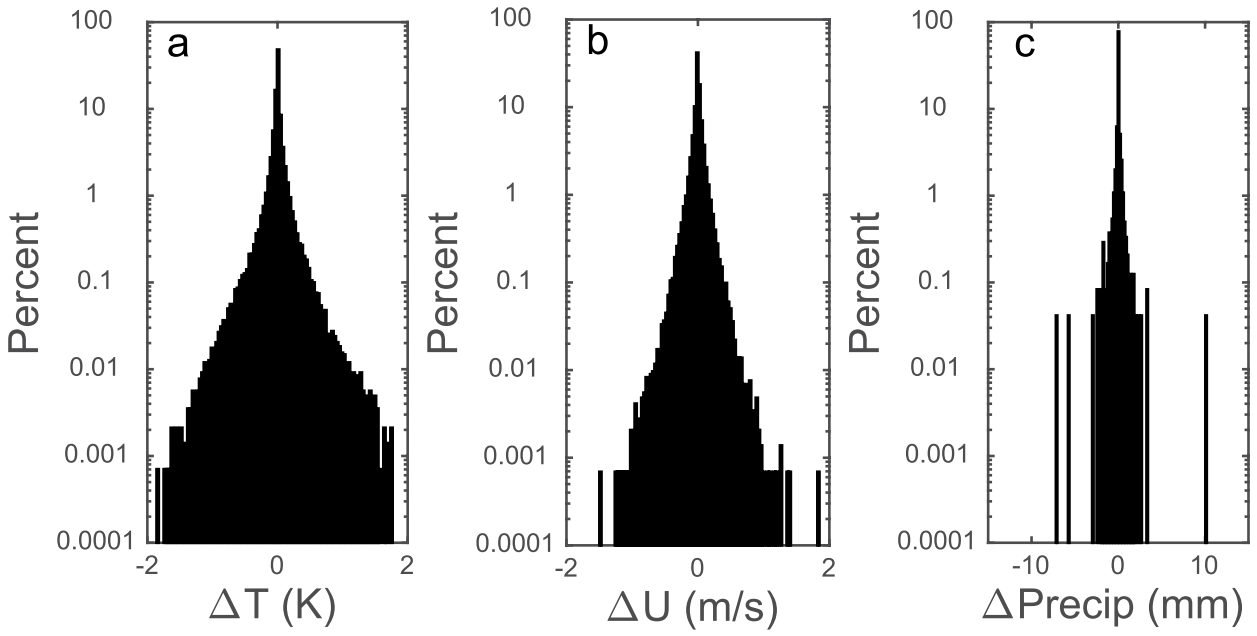

Fig. 3. Differences between gridpoint values of two Sandy simulations: (a) temperature, (b) zonal wind, and (c) precipitation valid 0000 UTC 28 Oct 2012 after 12 h. WRF executables differ only in a single compiler optimization flag. All grid points in the 3D volume are included.

they make use of any resources locally available to individual researchers. In general, identical compilers and operating systems are not available to collaborating researchers, except when all participants are working on the same community computing resource and in the same environment. Numerical reproducibility even within collaborating groups is not straightforward. With containers, collaboration among groups of scientists that need to work on the same code, or produce parallel simulations with only small variations, is immediately enabled. Examples include ensemble forecasts run on distributed and inhomogeneous platforms by multiple people in different locations, model developers from different institutions who are working to improve a physics scheme, or researchers collaborating to better quantify an energy budget within the model.

Educational activities clearly benefit from containers. Hands-on exercises are a valuable part of any student learning experience. Containers give instructors control over what parts of the modeling system students will experience directly. For example, the WRF could be run initially as a "black box" and then intermediate steps be made accessible as students work through exercises. Software containers help standardize modules that can be easily shared among teaching colleagues with vastly different technology environments.

The broader science community has recently been placing a greater emphasis on reproducible research, and containers offer a key step toward reproducible NWP research. Published papers can cite a specific container version used in one part of a research flow (e.g., the model, set of analysis tools, or a data container with initialization files). As long as that container revision exists and is accessible, research consumers can reproduce the results. This provides not only an unprecedented level of openness in NWP research but also an easier way for researchers to build on published results. The resources in the next section offer one component of the suite of tools needed to enable fully reproducible science. Metadata and analysis methods also need to be tracked and made available. Tools for that exist, and they should be adopted by the NWP community as we go forward.

RESOURCES. This section provides links to online content, including documentation and open repositories for WRF-based containers. A basic knowledge of repositories, and access to GitHub and Docker Hub public repositories, are needed to pull source code or container images from the repositories. The following is a list of resources:

- Docker-WRF project home page: www.ral.ucar.edu/projects/ncar-docker-wrf
- Slack channel (user discussion forum): https://ncar-dockerwrf.slack.com/
- WRF–WPS container image: https://hub.docker.com/r/bigwxwrf/ncar-wrf/

- Static geography data container image: https://hub.docker.com/r/bigwxwrf/ncar-wpsgeog/
- Input data container for Sandy simution: https://hub.docker.com/r/bigwxwrf/ncar-wrfinputsandy/
- Input data container for Katrina simulation: https://hub.docker.com/r/bigwxwrf/ncar-wrfinputkatrina/
- NCL script container for producing images: https://hub.docker.com/r/bigwxwrf/ncar-ncl/
- GitHub repository Docker files and scripts to build images: https://github.com/NCAR/container-wrf

ACKNOWLEDGMENTS. Partial funding for this work where provided by National Science Foundation Awards ATM0753581/M0856145 to NCAR and 1450488 to the University of California, Santa Cruz. Sandia National Laboratories and LANL–UCSC Institute for Scalable Scientific Data Management (ISSDM) also contributed funding. Work at the University of North Dakota was funded by NSF ACI-1450168 (PI Gretchen Mullendore) and the North Dakota Space Grant Consortium. Cindy Halley-Gotway at NCAR produced the schematic in Fig. 1. Amazon Web Services is acknowledged for providing an educational resource grant used for some of these investigations.

REFERENCES

Baker, A. H., and Coauthors, 2015: A new ensemble-based consistency test for the Community Earth System Model (pyCECTv1.0). *Geosci. Model Dev.*, **8**, 2829–2840, doi:10.5194/gmd-8-2829-2015.

Felter, W., A. Ferreira, R. Rajamony, and J. Rubio, 2014: An updated performance comparison of virtual machines and Linux containers. IBM Research Rep. RC25482, 12 pp. [Available online at http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf.]

Jimenez, I., C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-Dusseau, 2016: Characterizing and reducing cross-platform performance variability using OS-level virtualization. *Proc. 2016 Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW 2016)*, Chicago, IL, IEEE, 1077–1080, doi:10.1109/IPDPSW.2016.97.

Lorenz, E. N., 1963: Deterministic nonperiodic flow. *J. Atmos. Sci.*, **20**, 130–141, doi:10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.

——, 1969: The predictability of a flow which possesses many scales of motion. *Tellus*, **21A**, 289–307, doi:10.3402/tellusa.v21i3.10086.

Skamarock, W. C., and Coauthors, 2008: A description of the Advanced Research WRF version 3. NCAR Tech. Note NCAR/TN-475+STR, 113 pp., doi:10.5065/D68S4MVH.

Thomas, S., J. P. Hacker, M. Desgagné, and R. B. Stull, 2002: An ensemble analysis of forecast errors related to floating point performance. *Wea. Forecasting*, **17**, 898–906, doi:10.1175/1520-0434(2002)017<0898:AEAOFE>2.0.CO;2.