SANDIA REPORT

Unlimited Release
Printed October 2018

End-to-end Provenance, Traceability, and Reproducibility Through "Palletized" Simulation Data

Gerald F. Lofstead, Andrew J. Younge, Joshua Baker

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

Unlimited Release
Printed October 2018

End-to-end Provenance, Traceability, and Reproducibility Through "Palletized" Simulation Data

Gerald F. Lofstead, Andrew J. Younge, Joshua Baker

Abstract

Trusting simulation output is crucial for Sandia's mission objectives. We rely on these simulations to perform our high-consequence mission tasks given our treaty obligations. Other science and modelling needs, while they may not be high-consequence, still require the strongest levels of trust to enable using the result as the foundation for both practical applications and future research. To this end, the computing community has developed workflow and provenance systems to aid in both automating simulation and modelling execution, but to also aid in determining exactly how was some output created so that conclusions can be drawn from the data.

Current approaches for workflows and provenance systems are all at the user level and have little to no system level support making them fragile, difficult to use, and incomplete solutions. The introduction of container technology is a first step towards encapsulating and tracking artifacts used in creating data and resulting insights, but their current implementation is focused solely on making it easy to deploy an application in an isolated "sandbox" and maintaining a strictly read-only mode to avoid any potential changes to the application. All storage activities are still using the system-level shared storage.

This project was an initial exploration into extending the container concept to also include storage and to use writeable containers, auto generated by the system, as a way to link the contained data back to the simulation and input deck used to create it.

Acknowledgment

The authors would like to acknowledge the help of Sylabs for learning enough internals of the Singularity system we extended in order to explore the ideas.

Contents

List of Tables

Nomenclature

Container The popular name describing using the Linux kernel concept of cgroups and namespaces to create a limited, private environment for an application to run. Docker [8] is the most common container system variety, but Singularity [3] offers the security and other features Sandia needs to meet our mission requirements.

Data Pallet Our coined term to describe a container that is created dynamically at runtime to hold any generated data. This container, unlike typical containers, is writeable persisting any changes when the container is unmounted from the system.

Chapter 1

Introduction

The existing paradigm for data management and traceability relies on external workflow management tools, such as the Sandia Analysis Workbench (SAW), to track artifacts. Unfortunately, these approaches have been problematic because the workflow engines do not manage the artifacts themselves as much as try to keep track of them. If a user or system migrates data, then SAW has to figure out what happened. This project is providing a low-level, first step for a viable fix for this problem. By encapsulating the data (and other artifacts) into a system-generated wrapper, we can use the unique ID for each wrapper as a tag and add that tag to all associated wrappers. More directly, we can wrap the application in a container with a container ID, the input deck in a container with a container ID, and then generate storage container(s) at runtime and add the application and input deck container IDs as annotations to the storage container. Then, by inspecting the storage container, it is simple to find the matching container for the application and input deck based on their associated container IDs. There is no question of whether or not these are the matching components or not. Further, since the annotations are in the wrapper and it is not possible to move the component without moving the wrapper (without explicit, difficult work), these identifiers move with the component making identifying both the ancestry and dependency webs straightforward.

The potential impact of this work is enormous. For the mission, having An established provenance record of what source and configuration created a data set offers a guarantee we cannot manage today. The problematic approaches being pursued both historically and today to address this issue are divorced from the underlying system that users and system processes will manipulate outside the management system causing links to get stale and broken. Only with an approach like this can we have full confidence that our ModSim analysis can be trusted and be traceable back to the original source code and people that created it (and documents specifying what the person should have developed even). For the broader science community, reproducibility, or confidence that the results presented are sufficiently documented that where they came from and how they were created is sufficiently documented to trust and potentially recreate them, is a key requirement for something to be a science. This system enables automatic tracking and use of support information without requiring a separate tool–the information is all inherent in the artifacts themselves.

The rest of this document is divided into the following sections. First is a short look at related work in Section 2. Next, is a description of the design in Section 3. Some

information about the measured overheads in this approach are included in Section 4. Finally, Conclusions and Future Work are described in Section 5

Chapter 2

Related Work

Existing work that attempts to do similar things does not really exist. This idea and approach are wholly new. However, there are some systems and work that are doing some more distantly related things.

Workflow systems, such as the Sandia Analysis Workbench [4], Pegasus [1]/DAGMan [7], Kepler [6], Swift [10], and many others all focus on providing a way to orchestrate a series of tasks to accomplish a goal. However, they offer little to no support for managing the artifacts, configurations for each component, and component versioning. The lack of system level support for these kinds of activities makes offering tools that provide them difficult to impossible.

The data pallets idea, at a surface level, is similar with various storage containers ideas. However, these approaches all focus on exposing a shared storage layer into the container context. None of them offer any sort of wrapper approach that could hold attributes linking data back to the antecedents.

Other approaches, if used correctly, have a potential to achieve the same goals, but with application modification. For example, ADIOS [5], NetCDF [9], and HDF5 [2] all offer arbitrary attributes inside the file. If a user were to explicitly add the container IDs, they could achieve a similar goal. However, that would require that the container IDs be apparent in the application's running context and that the application code be modified to include writing such annotations into the file. The Data Pallets approach eliminates all of those requirements by adding the wrapper automatically just by virtue of running the application in a container and the annotations are generated the same way. No changes the the application or how it runs will be required.

Chapter 3

Design

The basic design for this proof of concept used an initial idea that proved too difficult and a fallback that was achievable.

The initial design idea was to intercept a call to the system command "mkdir", the command to create a folder in storage, and use that as the entry point for creating a new container on the fly and mounting it. Because of how containers work, there is no straightforward intercept point for this command that we could discover given the limited time frame.

The revised design idea was to create a single container at start up that would house all written data and annotate that container with the container IDs and mounted directories for the rest of the runtime context. This proved much more achievable given the limited time and other complications.

Design and Implementation Challenges

To accelerate the idea exploration, we chose to leverage the Singularity system. It already offered a way to mount an existing container as "writeable", but did not offer a way to generate that on-demand nor to annotate it with any sort of context information. The primary difficulty was twofold. First Singularity 2.x and earlier is a mixture of C, Python, and some Go code. However, some difficulties in using Python and the extra complexity of having 3 languages prompted the team to shift to nearly all Go with a small bit of C code used for starting up. This transition was partially underway as this project started and only finished a few days before the end of the funding period. The difficulties were many. First, the use of Go's dependency system did not work properly through the Sandia proxy and with the self-signed security certificate. After consulting internal Go experts, it was deemed that the only way to make it possible to build the 3.0 code base was to be on a machine outside the SRN so that it could properly check and download the dependencies prior to building. Once built, we could email transfer and scan all of the code for potential issues and use on our SRN machines. This was clearly far from ideal.

Second, the singularity container file format that supported annotations and writeable containers only worked with the 3.0 release. This required us to work largely off the SRN

and do mostly unbuilt code exploration on the SRN. At the close of the funding period, the Singularity code based had migrated to a newer dependency model that works properly through the Sandia proxy and self-signed certificates.

In spite of these challenges, the project was able to develop an initial hand-coded demonstration and measure some of the salient characteristics that would help determine if the overheads are viable and if it is worth pursuing in the future.

Design and Implementation Details

Using the fallback position of building a new container at start up proved mostly viable. In the code, the place where all of the specified containers and directories are mounted is in the src/runtime/engines/singularity/container.go file. Initial attempts to add the container creation here were thwarted by a partially broken build system that recognized changes in this file required it to be rebuilt, but it used a cached copy instead avoiding actually rebuilding the code. Work fixing this part of the build process continues under the auspices of a different project.

Since the auto-build was not able to be made functional in time, a new container was created manually, mounted with the writeable flag, and the annotations of the context information was added to the writeable container–the data pallet.

Integration with Sandia Analysis Workbench (SAW)

A stretch goal for this project was to integrate with SAW to demonstrate that it could manage a fully containerized workflow. As a stopgap as problems were debugged by our external partners, we investigated how to integrate with SAW using the existing system. While this integration is basic, it is sufficient to be able to show that running containerized applications is both reasonable and straightforward. Details of how this worked are in the Section 4.

Chapter 4

Measurements

The time overheads for this approach are small, particularly in the context of the application runtimes. With time overheads on the order of around a second or less to create a container and mount it, these overheads are not included. Instead, we focus on the more important space overhead since that will persist for the data lifetime.

The second tests are focused on integrating with the Sandia Analysis Workbench. This test is simply a test of possibility rather than focusing on performance. Detailed are provided below.

Tested Configuration

All tests are performed on a Dell Precision 3420 SFF tower machine with a 1 TB SSD and 64 GB DRAM. It is running Ubuntu Lunix 18.04 and Singularity 3.0 alpha 2. The system was fully updated as of September 15, 2018 for all tools, compilers, and libraries.

The tested configurations are as follows:

- Empty Container - This will be the baseline overhead for all applications and input decks. Since we will store all of these artifacts long term, understanding this base overhead will reveal how much additional storage will be required for archiving.
- Basic Writeable Container - This is the size overhead for the generated data sets.
- Attributes Stream - Since each container image can contain multiple data streams, one will be the normal data contents and a second will be the JSON file containing the list of associated container IDs.

Integration With Sandia Analysis Workbench

The long-term goal for this work is to be able to deploy fully container wrapped applications and input decks into the SAW system and eliminate the need for SAW to attempt

Table 4.1. Full caption

Empty Container	32.2 KB
Writeable Container	704.5 KB
Attributes Stream	1.1 MB

to track generated artifacts and instead rely on the inherent annotations as the source of truth. The previously mentioned limitations with the current system that attempts to track artifacts without owning them is fragile with many exposed areas where the system can be broken either accidentally, on purpose, or by automated system processes.

To test this integration, a simple input deck was wrapped in a container, another container contained an application, and it generated a separate output. We chose to use the gnuplot application as there is a simple example using this provided as part of SAW.

By wrapping everything in a container, we changed the workflow to run the component by using the container runtime command rather than directly running the application. This simple change revealed the simplicity with which this change could be deployed with minimal to no impact on existing users. Further integration is left to future work.

Discussion

The empty container size is small enough to be used as a wrapper around any application. For an input deck, this size is likely larger than the input deck itself (excepting a mesh description), but still small enough to justify using it to wrapper around the input deck.

The writeable container size is much larger because the only writeable format is ext3 rather than squashfs, what is used for read-only containers. The reason ext3 is required is that the container system requires the underlying operating system to handle the writing operations into storage backed container file. The squashfs file system format inherently does not support writing into it except at creation.

The attributes stream overhead is surprisingly large. This would be the minimum size for each additional attribute stream added to a container image. For output data sets on the order of 10% of node memory, this is still a small enough overhead to be acceptable. Our nodes are moving to 128 GB as the base memory making this less than 1 % of the total volume written during the output.

Chapter 5

Conclusions and Future Work

While this project was fraught with unexpected challenges and delays, we were able to perform some basic measurements and demonstrate the most basic of system ideas in a controlled environment. This shows that the idea is possible and the overheads are reasonable given data size trends for our data.

For the future work, many steps will be investigated.

First, automating the container creation and mounting at startup for our data pallet is ongoing as the system setup and build environment is debugged. Once this is complete, the next step will be to dive deeper into the code and how containers work to attempt to discover if it is possible to mount a container later than startup or if that will require some fundamental change to the running container environment. If it is possible, then dynamically creating the container based on a "mkdir" command will be implemented localizing exactly the data being written as part of that output.

Second, issues surrounding when and how to unmount a writeable container need to be investigated. Currently, we are assuming that the system can flush cold data to storage without having to rely on the container context exiting. If this is not the case, then memory pressures are going to force changes such that we can discover how to unmount the created container to free needed memory.

Third, the SAW integration demonstrated the possibility of running containers. The next step for this to be usable and viable is to offer introspection into things like the JSON attributes stream, seeing a list of dependent containers, and getting a list of available containers to run from a local container "hub". These hubs offer a way to dynamically load a container as needed based on the container ID rather than having to manually pre-deploy all needed containers when running the application. Scalability issues with this are likely at the high end, but well understood and demonstrated techniques to manage this, such as distribution trees, can address the performance issues.

References

[1] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[2] Q Koziol and R Matzke. Hdf5–a new generation of hdf: Reference manual and user guide. *National Center for Supercomputing Applications, Champaign, Illinois, USA, http://hdf. ncsa. uiuc. edu/nra/HDF5*, 1998.

[3] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[4] Sandia National Labs. Sandia Analysis Workbench Next Generation Workflows, 2018. `https://gitlab.com/iwf/ngw`.

[5] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, New York, NY, USA, 2008. ACM.

[6] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.

[7] G. Malewicz, I. Foster, A.L. Rosenberg, and M. Wilde. A tool for prioritizing DAGMan jobs and its evaluation. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pages 156–168, 0-0 2006.

[8] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[9] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[10] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

DISTRIBUTION:

	MS	,
		,
1	MS 0899	Technical Library, 9536 (electronic copy)


Sandia National Laboratories