SANDIA REPORT

2018-XXXX
Unlimited Release
Printed August 2018

Embedding Python for In-Situ Analysis

Timothy M. Shead, Konduri Aditya, Hemanth Kolla, Daniel M. Dunlavy, W. Philip Kegelmeyer, Warren L. Davis IV

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

Sandia National Laboratories

2018-XXXX
Unlimited Release
Printed August 2018

Embedding Python for In-Situ Analysis

Timothy M. Shead
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1326

Konduri Aditya
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969

Hemanth Kolla
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969

Daniel M. Dunlavy
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1327

W. Philip Kegelmeyer
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969

Warren L. Davis IV
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1327

Abstract

We describe our work to embed a Python interpreter in S3D, a highly scalable parallel direct numerical simulation reacting flow solver written in Fortran. Although S3D had no in-situ capability when we began, embedding the interpreter was surprisingly easy, and the result is an extremely flexible platform for conducting machine-learning experiments in-situ.

Acknowledgment

This work was funded through U.S. Department of Energy Advanced Scientific Computing Research (ASCR) grant FWP16-019471. Sandia National Laboratories is a multi-mission 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.

Contents

1 Introduction

It is common in scientific computing workflows to record simulation states at regular intervals, typically spaced many timesteps apart. Because interesting events can manifest quickly, they are easily missed in the intervals between saved timesteps. Recording the global simulation state more frequently can mitigate this, but causes excessive runtime overhead for I/O [3]. Because interesting events are typically local, this also means that a majority of the disk resources consumed are wasted storing uninteresting or unchanging state.

We are conducting research on creating and using machine learning models in-situ to detect events of interest automatically, writing data to disk only when and where needed – in hopes of dramatically reducing overall I/O while ensuring that interesting events are never lost [8]. As an initial proof-of-concept, we developed machine learning algorithms to identify temporal and spatial events of interest using feature importance metrics computed from outputs written to disk by combustion (S3D) and climate (CAM5) simulation codes, both of which are written in Fortran.

Our algorithms were developed using the Python[1] programming language, as Python is widely used in machine learning, supporting popular tools such as Keras [5], Tensorflow [1], Theano [11], and Scikit-learn [12]. After establishing that our prototypes showed promise, our next step was to update the implementations so that they could be run in-situ. We wanted to continue using Python so that we could avoid the cost of porting our algorithms and their dependencies to a compiled language.

[1]https://www.python.org

2 S3D

S3D is a massively parallel direct numerical simulation (DNS) solver for multi-component reacting flows [4]. The code solves the governing conservation equations in a compressible flow formulation and uses finite difference methods to approximate derivatives of those equations. Spatial derivatives are computed with eighth-order accurate central difference schemes and tenth-order accurate filters to damp high frequency spurious noise. A six-stage fourth-order accurate Runge-Kutta scheme is used for time integration. The code is implemented in Fortran and can be run in parallel using the Message Passing Interface (MPI) [7].

3 Python Embedding

S3D is written in Fortran 90, with a few routines in Fortran 77. The control flow in S3D is organised in a *solve driver* that performs high level actions to setup the grid, compute the physics kernels, initialize the solution and perform the time advancement. When we began this work there were no in-situ capabilities in S3D, so our first step was to define an in-situ callback API that could be invoked from within the *solve driver*. We implemented a Fortran in-situ module that includes data members to determine the frequency (in S3D time steps) for performing in-situ analysis, a subroutine to read parameters for the analysis from an input file (*insitu.in*, described below) and a wrapper subroutine that invokes a C++ callback function. The wrapper subroutine passes metadata and pointers to relevant data from Fortran code to the callback function, including MPI rank, grid parameters, solution variables and time:

```
void insitu_callback(
    const int id,
    const int nx,
    const int ny,
    const int nz,
    const int nspecies,
    const double* u,
    const double* v,
    const double* w,
    const double* pressure,
    const double* temperature,
    const double* species,
    const int timestep,
    const double time)
{
        // ...
}
```

Within the callback, we used the Python C API[2], to embed an instance of the Python interpreter in the S3D executable and share simulation data between the compiled and Python environments.

The first step in this process is to initialize the Python interpreter and load the user code to be run in-situ. Since our callback will be called for every timestep in a simulation that might run for 10^6 or more timesteps, we use a static variable to ensure that the interpreter is only initialized once, and we cache a pre-processed representation of the user code that is loaded from a known filesystem location, so that it does not have to be re-loaded or re-parsed at each timestep:

```
static PyCodeObject* py_code = 0;
```

[2]https://docs.python.org/2/c-api/index.html

```
if(!py_code)
{
    const char* code = // Loaded from disk
    Py_Initialize();
    py_code = Py_CompileString(code, ...);
}
```

Once initialization is complete, converting the scalar metadata values passed to our callback into objects accessible from Python is trivial:

```
PyObject* py_id =
    Py_BuildValue("i", id);
PyObject* py_nx =
    Py_BuildValue("i", nx);
// ...
PyObject* py_time =
    Py_BuildValue("d", time);
```

(in Python, all values are instances of *object*, and a *PyObject** is its equivalent in the Python C API).

The bulk of the simulation state is stored using a set of 3-dimensional dense arrays containing velocity, pressure, and temperature fields for each cell of the simulation grid, plus one 4-dimensional dense array containing the concentrations of chemical species in each cell. These arrays are allocated by the S3D Fortran code and passed by-reference to our callback as pointers to the underlying memory. Converting them to something accessible from Python was trickier than the metadata, as Python does not have a builtin data structure for multi-dimensional arrays, and we wanted to avoid the overhead of copying the data, which would have slowed the simulation and doubled its memory footprint.

Happily, the *Numpy* [9] library provides dense, strongly-typed multidimensional arrays in Python and is widely used in scientific computing and machine learning, including all of the libraries used in our research. Numpy includes a wide variety of compiled operations including slicing, masking, filtering, statistics, linear algebra, and more, so that interpreted Python code using Numpy arrays can be nearly as efficient as its compiled counterparts. Crucially, Numpy arrays can be "wrapped" around existing arrays instead of allocating their own memory, and support both row-major and column-major ordering. This allowed us to convert the S3D arrays into Numpy arrays with near-zero overhead:

```
#include <numpy/arrayobject.h>
npy_intp dims[3] = {nx, ny, nz};
PyObject* py_pressure =
    PyArray_New(&PyArray_Type, 3, dims,
    NPY_DOUBLE, NULL, pressure, 0,
    NPY_ARRAY_FARRAY_RO, NULL);
```

Above, we pass a pointer to the block of memory containing the simulation pressure to *PyArray_New*, along with information about the number of array dimensions and its extent along each dimension. The *NPY_DOUBLE* flag specifies that the memory contains double-precision floating point values. The *NPY_ARRAY_FARRAY_RO* flag configures the new Numpy array to access the wrapped data using Fortran (column-major) ordering (since the underlying data was created by S3D), and prevents callers from modifying the wrapped data, so that in-situ code cannot alter the simulation state. Exposing the S3D simulation state as Numpy arrays is thus very efficient, while affording in-situ code the opportunity to use flexible, high level abstractions to index and access the data, passing it directly to libraries that use Numpy arrays as input.

With the simulation state converted to Python values, the remainder of our callback implementation determined how those values would be accessed (such as their naming and scope, which we will describe from the user's perspective below), and executing the user code. Because the in-situ code could contain runtime errors, we check for problems each time it is executed:

```
PyEval_EvalCode(py_code, ...);
if(PyErr_Occurred())
{
    PyErr_Print();
    throw std::runtime_error(
        "Uncaught Python exception.");
}
```

Here we print a backtrace with line numbers and throw a C++ exception if an error occurs. This causes the S3D process to exit immediately, which we considered preferable to a soft error that might be overlooked at runtime.

The complete in-situ Python implementation for S3D is less than 90 lines of Fortan 99 and 180 lines of C++ code. Using shared linking on a macOS workstation, the in-situ Python capability increased the size of the S3D executable on disk from 3.0MB to 3.1MB, while the size of the working set at runtime increases by 500KB. While real-world in-situ runtimes and resource consumption will vary according to the analysis performed, we found that simulation runtimes using a do-nothing script were virtually indistinguishable from runtimes with in-situ Python disabled. Overall, we felt that these were extremely reasonable overheads in exchange for the flexibility afforded by an entire general purpose programming language.

4 Runtime Interface

When the in-situ Python capability is enabled, the S3D executable looks for two additional files in its input deck at runtime. The first – *insitu.in* – is a text file that configures whether and how often the in-situ code is run. The second file, *insitu.py*, contains the Python source code to be executed in-situ. The code in insitu.py has the full generality of the Python language at its disposal, and can load any and all builtin, system, and external Python libraries needed to carry out an analysis.

4.1 Simulation State

The Python objects created by the C++ in-situ callback are exposed to in-situ Python code using a special global variable named *simulation*, which encapsulates the current simulation state using the following attributes:

id MPI rank of the current process.

nx, ny, nz size of the simulation grid.

nspecies number of chemical species in the simulation.

u, v, w read-only Numpy arrays with shape (nx, ny, nz).

pressure, temperature read-only Numpy arrays with shape (nx, ny, nz).

species read-only Numpy array with shape (nx, ny, nz, nspecies).

species_names list of species names.

time simulation time, in seconds.

timestep simulation timestep.

So an in-situ script to calculate and print the local maximum pressure could be as simple as:

```
print(simulation.pressure.max())
```

Note that *simulation.pressure* is a Numpy array, and *max()* is a method provided by the array. We will illustrate below how to calculate the global maximum pressure.

We implemented the *simulation* variable as a subclass of the Python *dictionary* type[3], so that it behaves in ways that typical Python users will expect, supporting features like indexing syntax and iteration:

[3]https://docs.python.org/2/tutorial/datastructures.html#dictionaries

```
print(simulation["pressure"].max())

for key, value in simulation.items():
    if isinstance(value,numpy.ndarray):
        print("%s max: %s"%(key, value.max()))
```

Attempts to modify any of the Numpy arrays in *simulation* will raise exceptions. Overwriting any of its attributes will have no effect on the simulation, and any new attributes added will be gone the next time the in-situ code is executed.

4.2 Persistent State

All variables created by the in-situ code are local variables that go out of scope (i.e. disappear) between timesteps. In this respect, the user's in-situ code is behaves like a function call, albeit without a function declaration or arguments. Because any non-trivial in-situ code will need to retain some state between timesteps, there is a second global variable – *state* – which we provide for that purpose. Like the *simulation* global variable, *state* is a subclass of a Python dictionary, so that it can be read from and written to using a mixture of attribute and indexing syntax. Unlike *simulation*, the contents of *state* are retained from timestep to timestep, so that in-situ code can use it as "scratch" storage for any information that needs to persist. This could include performing one-time setup when the simulation starts:

```
if "initialized" not in state:
    # Perform one-time setup here
    state.initialized = True
```

The following example uses *state* to write outlier (more than two standard deviations from the mean) pressure values to disk whenever the average pressure change from one timestep to the next exceeds a threshold:

```
mean_pressure = simulation.pressure.mean()
if "mean_pressure" in state:
    delta =
        state.mean_pressure - mean_pressure
    delta = abs(delta)
    if delta > threshold:
        cell_delta =
            simulation.pressure - mean_pressure
        cell_delta = numpy.abs(cell_delta)
        cell_threshold =
            2 * simulation.pressure.std()
        selection =
            cell_delta > cell_threshold
        outliers =
            simulation.pressure[selection]
        filename =
            "outliers-%d-%d.npy".format(
                simulation.id,
                simulation.timestep,
                )
        numpy.write(outliers, filename)
state.mean_pressure = mean_pressure
```

4.3 Inter Process Communication

Note in the previous example that we include the processor ID in the output filename. As a scalable parallel code, S3D can be run on many processors using MPI, with the simulation grid partitioned evenly among processes along each dimension. This subset of the global grid is what user code in the in-situ callback has access to, so that all of the examples seen thus far have been purely local computations. Using the processor ID for output ensures that processes don't overwrite each others' files, but this assumes that we will merge the output data in some post-processing step. There are many cases where it would be preferable to perform in-situ computation on global information instead.

There are numerous Python libraries that could be used for inter process communication; for our experiments we preferred to use MPI since we were already comfortable with the MPI communications model. The following example uses *mpi4py* [6] to store a timeseries containing the global maximum pressure across all processes. Note that MPI calls made from mpi4py are nearly identical to their C counterparts:

```
from mpi4py import MPI
if "communicator" not in state:
    state.communicator = MPI.COMM_WORLD
    state.max_pressure = []
max_pressure =
    state.communicator.allreduce(
    simulation.pressure.max(), MPI.MAX)
state.max_pressure.append(max_pressure)
```

4.4 File I/O

Even before moving our machine learning experiments in-situ, we found that a useful application of our new capability was as a kind of "universal file output" mechanism, supplementing the file I/O compiled into S3D with custom I/O we could write ourselves. This allowed us to write internal S3D state directly to disk using file formats that our postprocessing code could load without conversion, eliminating several steps in our workflow. As examples, we have used Python's builtin *pickle*[4] module to serialize complex, heterogeneous data structures, and Numpy's own array-centric I/O[5] to store individual arrays in binary format. To work with multiple large Numpy arrays and related metadata, the *h5py*[6] module provides efficient access to potentially huge HDF5 [10] files.

[4]https://docs.python.org/2/library/pickle.html

[5]https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.io.html

[6]http://www.h5py.org

5 Conclusions

Thanks to the simplicity and flexibility of the Python C API, we were able to easily integrate a Python interpreter within a Fortran code, and implement efficient zero-copy access to the simulation state. Using defacto-standard Numpy arrays to wrap that state means that in-situ Python code has access to a huge set of sophisticated scientific computing and machine learning tools. All that was needed to embed Python is a C or C++ callback function with access to the simulation state that is called at each simulation timestep.

We had complete control over how the simulation and in-situ code interact, and were able to craft an interface with a focus on simplicity and clarity that should feel comfortable for S3D and Python users alike. We particularly preferred the direct, minimally-intrusive nature of this approach to more elaborate in-situ platforms that have steeper learning curves and lack the generality of a general purpose programming language.

In the future we will be using a similar approach to embed Python in the other simulation code used for our machine learning experiments, the Community Atmosphere Model 5 (CAM5) [2].

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow - Large-Scale Machine Learning on Heterogeneous Distributed Systems. *Computing Research Repository*, 2016.

[2] R B Neale et al. Description of the NCAR community atmosphere model (CAM 5.0). *NCAR Tech. Note NCAR/TN-486 STR*, 2012.

[3] C Chen, Y Chen, K Feng, Y Yin, H Eslami, R Thakur, X H Sun, and W D Gropp. Decoupled I/O for data-intensive high performance computing. *IEEE International Conference on Parallel Processing*, pages 312–320, 2014.

[4] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001–32, January 2009.

[5] François Chollet. keras. Technical report, 2015.

[6] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[7] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[8] Julia Ling, W Philip Kegelmeyer, Konduri Aditya, Hemanth Kolla, Kevin A Reed, Timothy M Shead, and Warren L Davis IV. Using Feature Importance Metrics to Detect Events of Interest in Scientific Computing Applications . In *The 7th IEEE Symposium on Large Data Analysis and Visualization*, Phoenix, AZ, October 2017.

[9] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 12:2825–2830, October 2011.

[10] The HDF Group. Hierarchical Data Format, version 5, 1997.

[11] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[12] Stefan van der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)

v1.40

Sandia National Laboratories