Mega Python: Scalable Interlanguage Scripting for Scientific Computing

Justin M. Wozniak*‡, Timothy G. Armstrong†, Scott J. Krieder§,
Ketan Maheshwari*, Michael Wilde*‡, Ian T. Foster*†‡
* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
† Dept. of Computer Science, University of Chicago, Chicago, IL, USA
‡ Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA
§ Dept. of Computer Science, Illinois Institute of Technology, Chicago, IL, USA

***Abstract*—Scripting languages such as Python and R have received wide adoption as tools for highly productive development of scientific software because of the power and expressiveness of the languages and available libraries. However, deploying scripting languages on large-scale production systems such as the IBM Blue Gene/Q or Cray XE6 is a challenge due to operating systems limitations, interoperability challenges, parallel filesystem overheads due to the small file system accesses common in scripted approaches, and other issues. In this work, we present a new approach based on Swift to integrate high-level languages such as Python, R, Tcl, and the shell with native code developed in C, C++, and Fortran, through the use of the library interfaces to the script systems. In this approach, Swift handles data management, movement, and marshaling among distributed-memory processes without direct user interaction with messaging libraries (such as MPI).**

I. Introduction

Many modern scientific applications and tools are built by using a variety of languages and libraries. These complex software products combine performance-critical libraries implemented in native code with high-level functionality expressed in rapidly developed and modified scripting languages. Additional specialized features may be used for concurrency, I/O, the use of accelerators, and other features. A wide range of application domains have used these development techniques, including materials science, protein analysis, and power grid simulation.

Each of these applications and tools follows a common software development pattern. First, a native code library is built or repurposed for the core processing. Then, a scripted toolkit is built around the core library or program. Such "wrapper scripts" could be developed in shell scripts, Python, Tcl, or other tools. Then, when additional scalability is required, scripts are developed to deploy the application in some distributed computing model such as MPI, Swift, or custom wrapper scripts that submit jobs to a scheduler such as PBS. Swift [1] is a programming language and runtime designed to ease the software development pattern described above. Swift has a well-defined concept of wrapper scripts, the ability to coordinate calls to tools through its programming model, and built-in support for many schedulers and data movement protocols.

The latest implementation, Swift/T [2], generates an MPI program from the Swift script and provides tools to run that program on varying scheduled resources. The Swift/T framework supports direct calls to native code through library loading and access. However, as described above, modern scientific applications are not built with native code alone, but with scripts and scripting interfaces to core libraries. Thus, to ease the coordination of calls to tools in the Swift programming model, we wish to support direct calls to script code without calling external programs or forcing the user to master complex linking techniques.

In this work, we report on new features in Swift to support direct calls to Python, R, and Tcl. These features could easily be extended to other script languages in a similar pattern in the future. These features allow Swift scripts to orchestrate distributed execution of code written in a wide variety of languages, currently including C, C++, Fortran, Python, R, Tcl, and the shell. Any external programs may be called through the shell-based technique.

The method presented here is a more approachable software development technique for distributed-memory computing than are traditional techniques. Using MPI, the developer could write MPI code in C and call to an application component script. In this technique, the user would have to manage the call to the script, possibly using an internal API specific to that language. Application data would have to be marshaled to and from the component and among processes in a tedious manner. The developer would have to define a progress model and manage load balancing and other distributed computing challenges. Alternatively, the developer could try a scripting language-specific MPI implementation, which might ease some but not all of the described challenges. Additionally, this would limit the number of languages that could be used; it is unlikely that communicating among MPI processes in multiple languages would work as desired.

In our method, the developer starts with a Swift script that describes the calls to application components in a convenient syntax. Data is passed from Swift to language-specific components over MPI without user marshaling. Multiple components written in different languages may be brought together. Progress and load balancing are managed by the Swift runtime. Overall, the approach provides a coherent programming model, allows for compatibility among multiple languages, provides high scalability, and is compatible with advanced

architectures such as the Cray XE6 and Blue Gene/Q.

The remainder of this work is organized as follows. In Section II, we describe relevant application models in detail. In Section III, we describe the architecture of Swift/T and in Section IV we describe the interlanguage features that are the focus of this paper. In Section V, we present performance results from Swift/T running on Blue Gene and Cray systems. In Section VI we describe related work and in Section VII we offer concluding remarks and a brief glimpse at future work.

II. Applications

In this section, we describe some interlanguage applications that serve as motivating examples for our work.

A. NeXus: Storage and processing for materials science

NeXus [3] is a file format for X-ray, neutron, and muon scattering data based on HDF [4]. By standardizing some of the metadata used in the HDF file, the X-ray scattering data is easier to distribute and analyze by different research groups. NeXus data is typically reused heavily for coordinate transformation, analysis, and visualization. The data is stored in large (up to 30 GB) multidimensional (up to 4D) arrays. Fragments of the underlying data variables may be processed concurrently in a task-oriented model.

NeXpy [5] is a Python-based GUI and scripting suite to operate on NeXus data. Many important data transformation, analysis routines, and visualization capabilities are made available through its scripted Python API. NeXpy users can easily rotate NeXus scattering data and perform common visualization operations. Underlying numerical operations are performed with NumPy [6], a numerical library for Python.

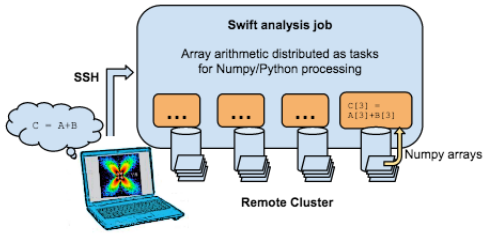

Figure 1. Parallel data analysis via Swift/NumPy processing in cluster.

Recent work in scaling NeXpy processing to larger datasets (30 GB) has motivated us to leave the bulk data on the remote cluster for processing and to transfer only the resultant plots back to the user workstation. This approach requires that the array processing be broken into tasks for distributed processing via Swift tasks. However, it is highly desirable to reuse as much of the existing NeXpy codebase as possible, so relevant functionality is called as Python tasks from Swift. To accomplish this effectively, we need interlanguage support to pass NeXus/NumPy data from Swift to Python and back.

B. OOPS: Protein simulation

The Open Protein Simulator (OOPS) is an interlanguage implementation of a protein folding simulation code [7]. At its core OOPS is built on the C Protein Folding Library, also known as protlib, a minimal library of C functions and data structures intended for generating folding simulations of proteins. It is based on a modular architecture that allows the use of different sampling algorithms and energy functions. In addition, protlib provides a Python interface to use within the Bio.PDB module of the Biopython library. OOPS reads a collection of protein configuration files through Biopython and makes core calls to protlib. OOPS leverages Swift to run across many nodes for a large scale protein folding simulation solution. The OOPS interlanguage software stack is shown in Figure 2.

Recent work aims to enable OOPS to make use of hardware accelerators including NVIDIA GPUs, and the Intel Xeon Phi [8]. In this effort, the OOPS libraries will be refactored to allow Swift to manage calls to the Python, C, and GPU-based features. This arrangement will make use of the advanced performance available on the GPU while also using higher level features in the Python libraries.

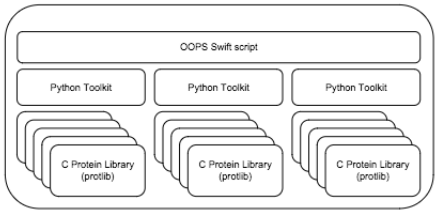

Figure 2. Component architecture of OOPS application.

C. Electrical power price analysis

The electrical power prices in a region are a result of combination of many stochastic and temporal factors, including variation in supply and demand due to market, social, and environmental factors. Evaluating the feasibility of future generation power grid networks and renewable energy sources requires modeling and simulation of this complex system. In particular, the power grid application described here is used to statistically infer the changes in the unit commitment prices with respect to small variations in random factors.

The application involves running a stochastic model for a large number of elements generated via a three-level nested `foreach` loop, as shown in the Swift code snippet below:

```
int nS[] = [10, 100, 1000, 10000, 100000];
foreach S, idxs in nS {
  sample0 = gensample(wind_data);
  obj[idxs] = ampl(sample0);
  foreach B, idxb in [10:40:10] {
    foreach k in [0:B]{
      sample1 = gensample(S, wind_data);
      obj_l[idxs][idxb][k] = ampl_L(sample1);
      sample2 = gensample(S, wind_data);
      obj_u[idxs][idxb][k] = ampl_U(sample2,
                                    obj[idxs]);
}}}
```

In this code, `gensample()` is not pure- it uses random numbers produced by the underlying task, producing different samples each time. Then, the numerical algorithm is run to compute lower (L) and upper (U) bounds, which converge for large enough S. (Only the upper bound computation consumes the output of function `ampl()`).

A moderate sample size of five samples can generate hundreds of thousands of Python calls. Each application call makes call to the Python-implemented sample generation (`gensample()`) and AMPL models making it an interlanguage implementation spanning Python and AMPL interpreters, as depicted in Figure 3.

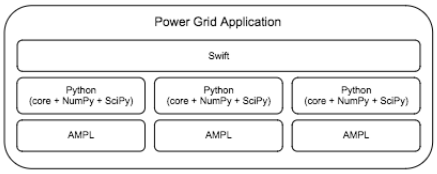

Figure 3. Electrical power price analysis application components.

D. DISCUS: Crystal structure scattering simulation

DISCUS [9] is a Fortran-based program for computing diffuse scattering of a simulated input crystal structure. DISCUS allows a user to run artificial experiments on crystal structures and produce outputs analogous to those of real experiments, for example the images that would be produced from an X-ray scattering experiment.

A recent effort involved using DISCUS to fit input parameters (crystal configurations) to experimental data. The output of a simulated DISCUS experiment is compared for fit against results of a real experiment, allowing the accuracy of the input parameters to be gauged. An evolutionary algorithm is used [10] to iteratively adjust the parameters to improve the fit, as shown in Figure 4.

Two levels of parallelism have been identified in this compute-intensive process. First, the DISCUS run itself can be improved through the application of thread parallelism in OpenMP. Second, the DISCUS runs can be called concurrently. Initial efforts by the DISCUS team ran into development issues when attempting to fit complex DISCUS parameter data into an ad-hoc master-worker parameter passing scheme. This is an ideal use case for the work presented in this paper because Swift includes a load balancer in a scalable master-worker scheme with multiple masters, along with flexible interlanguage data handling.

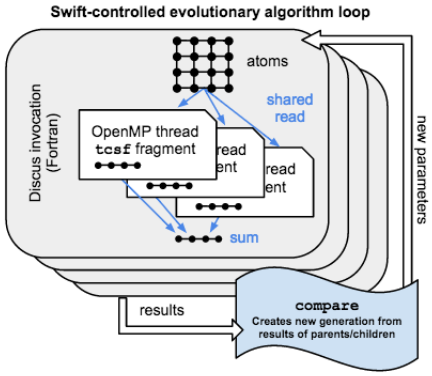

Figure 4. Multi-level parallelism in DISCUS.

E. Generic application models

With these real-world applications in mind, we have a basis to describe a general model for interlanguage scientific applications.

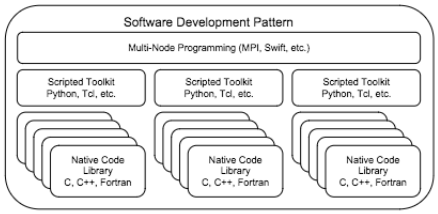

Figure 5. Existing software model supported by Swift.

In the model shown in Figure 5, existing application components of native code libraries wrapped in scripted tools are then wrapped at a higher level by Swift. This approach allows the reuse of application logic while providing concurrency at the Swift level.

A subtle change is introduced in the model shown in Figure 6. In this model, scripting language components are brought close to the Swift level as a result of tight interlanguage support by Swift features and the performance boost due to linking to the scripting language library (as opposed to calling the script interpreter as an external program).

III. Architecture

In this section, we provide background on the Swift language, describe the Swift/T architecture for reference and discuss how Swift/T calls scripted application components.

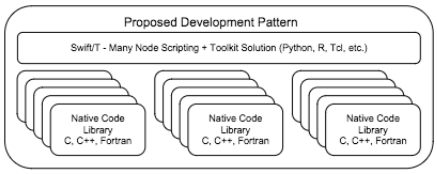

Figure 6. New software model- scripting tools (Python, R, etc.) are integrated closely with the Swift script.

A. Swift language

Swift is a scripting language with C-like syntax with pervasive, automatic concurrency built into the language. Concurrency is achieved through dataflow processing, in which progress depends on the availability of input data, not statement ordering. For example, in the code fragment

```
1 | int x;
2 | x = f(3);
3 | int y1 = g(x,1);
4 | int y2 = g(x,2);
```

the declaration `int x;` creates a *future* `x`. Subsequent function calls to `g()` block until a value is stored in `x`. When `f()` completes, both calls to `g()` are eligible to run concurrently on different processors.

Massive concurrency can be achieved in Swift with relatively little code. For example, in the code fragment

```
1 | foreach i in [0:9] {
2 |   int t = f(i);
3 |   if (g(t) == 0) { printf("g(%i)==0", t); }
4 | }
```

the `foreach` loop executes each loop body for a unique value of `i` from 0...9 concurrently. Each execution of `f()` may be run concurrently, but each `g(t)` is blocked on the corresponding `f(t)`. The code implies the dataflow dependencies shown in Figure 7,

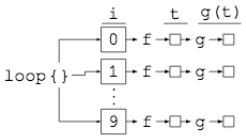

Figure 7. Diagram of implicit dataflow of Swift loop.

where several parallel pipelines of tasks are present. Swift will construct and execute these pipelines in parallel on any available resources.

In the Swift model, bulk user computation is performed in *leaf tasks*: user code outside of Swift, such as libraries or external programs. These are load-balanced between available processors by dispatching tasks on demand. If `f()` and `g()` are compute-intensive functions with varying run times, the asynchronous, load-balanced Swift model is an excellent fit.

B. Swift/T runtime

Swift/T [2] is a reimplementation of the Swift/K [1] language.

Swift/K excels at distributed, grid, and cloud computing, and offers wide-ranging support for schedulers (PBS, LSF, SLURM, SGE, Condor, Cobalt, SSH) and data transfer, fault tolerance, and other features useful for that environment. K indicates that the language is implemented atop the Karajan workflow engine.

Swift/T is designed for high-performance computing at the largest scale, offering a runtime based on the Asynchronous Dynamic Load Balancer (ADLB) [11]. T indicates that the key language features are implemented by the Turbine dataflow engine [12]. In this implementation of Swift, the Swift script is translated into a runtime framework based on the C-based ADLB and Turbine libraries, which evaluate Swift semantics in a distributed manner (no bottleneck).

The Swift/T architecture is diagrammed in Figure 8. Each process operates as an engine, ADLB server, or worker. Engines carry out Swift logic, creating leaf tasks for execution. ADLB servers, shown as an opaque subsystem, distribute tasks to workers, which execute user work (such as `f()` and `g()` in our example above). Typically the vast majority of processes (99%+) are designated as workers. The engine and server processes are called *control processes* and collectively orchestrate script execution.

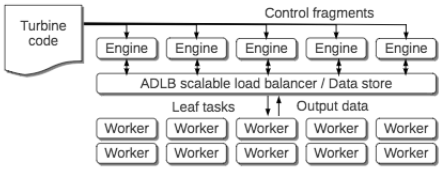

Figure 8. Swift/T runtime architecture.

IV. Swift interfaces to various languages

Swift/T has multiple new methods for calling to user code that have not been reported previously. In this section, we consider these in detail.

A. Calling the shell

In Swift/K, leaf tasks were intended primarily to be developed as calls to `qsub` on remote systems. Following the monolithic MPI model, however, Swift/T interacts with the shell as a local library, because the Swift/T worker is just another process in the MPI run. Interaction with the shell is defined in Swift using function annotated with `app` for "application".

Consider the following Swift function definition and call:

```
1 | app (file o) prog1 (string S[][], int i) {
2 |     "/bin/prog1" (S[0]) "--" (S[1]) i o;
3 | }
4 | ...
5 | file f<"output.txt"> =
6 |   prog1([["-v"], ["foo","bar"]], 42);
```

User program `/bin/prog1` is made available to Swift as function `prog1()`, with a type signature that indicates it accepts a two-dimensional array of strings and an integer, and produces one file as output. (In Swift, files may also be used as part of the dataflow structure.) Elsewhere in the Swift script, a file `f` is defined as the output of `prog1` and mapped to a location in the filesystem, `output.txt`. The user passes `prog1` a two-dimensional string array literal and an integer literal. Following the `app` definition, Swift converts these variables to shell command

```
1 | /bin/prog1 -v -- foo bar 42 output.txt
```

Swift does not attempt to open `output.txt` itself; it assumes that the user program will create that file.

This functionality packages multiple features that allow the expression of complex interlanguage issues between Swift and the shell.

First, note that the shell command line is unstructured compared with the ability of Swift to represent structured data. All command lines must fit the C-based `main(int argc, char** argv)` model. Thus, Swift data structures are flattened into simple strings for the command line. Note, however, that the ability of Swift to evaluate arbitrary code while constructing the command line (here, indexing into the array `S`) allows clean separation of flags and arguments, as is conventional, with the use of the `--` symbol.

Second, note that the shell command line does not support typed data. Thus, Swift converts various types to strings; in this case, an integer and a file variable are placed on the command line. Since Swift does support types, including subtyping on `file` to create specific file types, many type errors common in shell scripting are easily avoided.

B. Calling Tcl

The Swift/T compiler (STC) translates user Swift code to a representation (Turbine code) that uses the Turbine, ADLB, MPI, and user libraries, all of which are written in C. While STC could generate C code, we desired a compiler target with the following properties: 1) A straightforward way to ship code fragments through ADLB for load balancing and evaluation elsewhere, 2) A textual, easily readable format, and 3) A runtime that did not require the user to run the C compiler to avoid complexities on advanced systems. Thus, we chose Tcl to represent Turbine code, and made use of the ease of calling C from Tcl to bind the system together.

Since Swift/T runs on Tcl, calling from Swift to Tcl is the most advanced interlanguage feature in Swift/T. Consider the Swift code fragment

```
1 | (int o) f(int i, int j)
2 | "my_package" "1.0"
3 | [ "set <<o>> [ f <<i>> <<j>> ]" ];
4 | ...
5 | int x = f(2, 3);
```

In this code, Tcl procedure `f` is made available to Swift with the given signature. When inputs `i` and `j` are available, the Tcl code (line 3) is executed. Tcl package `my_package 1.0` is loaded on the assumption that `f` will be found in that package. The Swift/T runtime supports user additions to `TCLLIBPATH` so that arbitrary Tcl code may attached to a Swift/T run.

Interlanguage operation is supported by 1) inserting dataflow semantics to the interface between Swift/T and Tcl and 2) automatic type conversion. The Tcl code on line 3 cannot execute until inputs `i` and `j` are set and transmitted to the worker on which the code will be executed, and storage for output `o` has been allocated. This code is automatically inserted into the compiler output by STC, and is hidden from the user (by default). The programmer provides a template for the Tcl code. Double angle brackets <<·>> indicate that a variable should appear in that location. Swift/T variables are automatically converted to the appropriate Tcl types, which are oriented toward string representations.

The ease of interlanguage operation here offers multiple beneficial features to Swift/T development and application users. First, the ease of exposing simple Tcl snippets to Swift allowed for the rapid development of Swift builtins such as `printf()`, `strcat()`, etc. Many Tcl features can easily be brought into Swift this way. Second, Swift users often express a desire to mix dataflow programming with short fragments of imperative code. This is easily done by extending the Tcl fragment on line 3 to a multiline script snippet, using the Swift multiline string syntax. Certain arithmetical or string expressions may be easier to perform in Tcl than in Swift, especially for experienced Tcl or shell programmers. Third, existing components built in Tcl can easily be brought into Swift using Swift support for Tcl packages. Fourth, the strength of Tcl support for calling native code is easily brought to Swift as well, as described in the following subsection.

C. Calling native code

A primary goal of Swift/T is to speed the development process for scaling existing codes in compiled languages (C, C++, Fortran) to high-performance systems. Thus, good support for calling these languages is paramount. Tcl provides good support for calling native code, and good tools such as SWIG are available. This approach has demonstrated the ability to successfully call native code in many applications, including applications that may be expressed as MPI libraries [13].

In order to call into an existing native code program from Swift, the following steps must be followed. First, the user identifies the key functions to be called. Simple types (numbers, strings) must be used to ensure compatibility with Swift. Second, the program is compiled as a loadable library - any use of `main()` must be removed through conditional compilation. Third, the library headers are processed by SWIG to generate Tcl bindings for the C/C++ functions; in the case of Fortran, a C++-formatted header is first created with FortWrap, then processed by SWIG. Fourth, the user writes Swift bindings for the generated Tcl bindings as described in the previous subsection. Fifth, a Tcl package is constructed containing the native code library and any additional Tcl scripts that the user desires to include. Figure 9 illustrates the process of binding a C code with Tcl using SWIG. The functions in object `afunc.o` become callable from within Swift/T code.

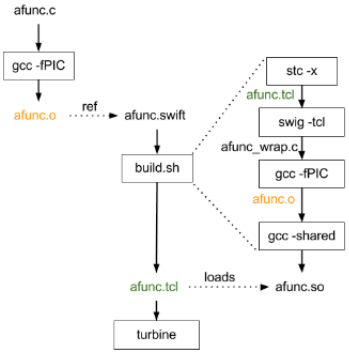

Figure 9. SWIG providing Tcl bindings for C functions callable from Swift/T.

The interlanguage complications here are more challenging than that in the Tcl case because more language considerations must be taken into account. Our approach has been to delegate complexities and conventions to SWIG, since that is a general purpose tool (i.e., learning SWIG has broader utility than learning a Swift-specific tool). Thus, type conversion conventions are delegated to SWIG conventions.

In addition to simple types, scientific users of native code languages often desire to operate on bulk data in arrays. The Swift approach to these is to handle pointers to byte arrays as a novel type: `blob` (binary large object). The Swift/T runtime handles blobs in a similar manner to strings, but with appropriate handling for binary data. This allows users to write dataflow scripts that operate on C-formatted strings and arrays, contiguous binary data structures, and even multidimensional Fortran arrays.

SWIG supports operations functions that consume and produce pointers as represented by Tcl variables. Thus, Swift/T provides a small library called `blobutils` to handle transmission of the Swift/T blob type to raw pointers compatible with SWIG. Type conversion routines are provided to handle many common cases. For example, SWIG will not automatically convert `void*` to `double*` - `blobutils` provides tools to handle the simple but myriad interlanguage complexities found when operating on binary data.

D. Calling Python or R

As described above, many modern scientific applications have key components or interfaces built in Python, R, or other dynamic script languages. Previous workflow programming systems call external languages by executing the external interpreters. As described previously, this is undesirable for Swift/T because at large scale, the filesystem overheads are unacceptable. Additionally, on specialized supercomputers such as the Blue Gene/Q, launching external programs is not possible at all.

Our approach, based in Swift/T, treats the external interpreters for Python and R as native code libraries. Thus, the complexity of calling them is reduced to the complexity of calling a C library from Swift/T, which was addressed in the previous section. First, a Tcl extension for each language was constructed. (These could conceivably be reused by non-Swift developers who simply desire to call Python or R from Tcl.) Then, a Swift/T leaf function was written that evaluates fragments of code.

In the Swift model, each task is started without state- only the well-defined Swift inputs are available. When calling into an external interpreter, however, old state from the previous task could be available and cause confusion or debugging issues (this is not a security issue, since all of this state is inside the Swift/T MPI run). One approach is to finalize the interpreter at the end of each task and reinitialize when the next task is started, thus clearing any state. We did not take this approach because of to concerns about performance and possible resource leaks. Thus, users must cleanly unset or overwrite any reusable data in their use of the external interpreters.

E. Calling Python and R

In addition to supporting rapid, concurrent calls to scripted application components, Swift/T also supports the ability to run tasks from multiple languages in a single script, and pass data among them. This allows the unique opportunity to combine these languages in large scale applications.

In the case study described here, we construct several matrices and find the biggest parallelepiped volume via NumPy and R. First, we use NumPy to create NumPy arrays and perform simple matrix arithmetic. Then, we compute determinants in parallel with NumPy. Next, we reduce to the maximal determinant using R. This basic procedure is depicted in Figure 10.

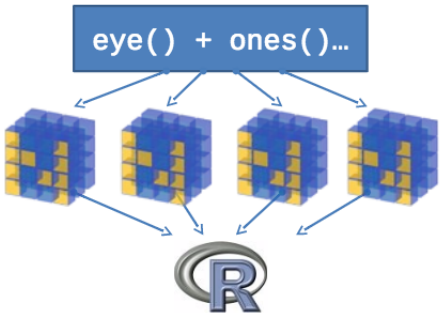

Figure 10. Graphical depiction of algorithm combining NumPy and R.

The Swift definition of NumPy features is packaged in a Swift header file for reuse. This Swift code performs minor transformations to convert the operation on Swift data to

Python code that uses NumPy, then evaluates the string in the Python interpreter and returns the result. A representative function, `eye(int N)`, is shown in Figure 11. This function simply creates a code fragment to call the NumPy function `eye()`, which returns I_N, and returns the result. The matrix is represented in Swift as a string.

```
1  global const string numpy = "from numpy import *\n\n";
2  typedef matrix string;
3  (matrix A) eye(int n) {
4    command = sprintf("repr(eye(%i))", n);
5    code = numpy+command;
6    A = python(code);
7  }
```

Figure 11. Fragment of Swift header to provide NumPy features.

The Swift code for the parallelepiped application is shown in Figure 12. First, the NumPy library is imported (line 1). Second, each matrix is constructed as $A = I_N * i + 1$, then A[2,0] is set to a different number for each iteration (lines 5-8). Third, the determinants are computed (concurrently), made positive, and stored in a Swift array of `float`. Finally, the R function `max()` is used to obtain the maximal value.

```
1  import numpy;
2  //  Define our collection of determinants:
3  float dets[];
4  foreach i in [1:U-1] {
5    //  For U, i, construct a matrix via Numpy:
6    A = matrix_add(matrix_scale(eye(N), itof(i)),
7                   ones(N));
8    B = matrix_set(A, 2, 0, (U-i+1)**3);
9    //  Obtain its determinant via Numpy:
10   v = determinant(B);
11   //  Store the determinant in a Swift array:
12   dets[i] = abs_float(v);
13   printf("dets[%i]=%.2f", i, v);
14 }
15 //  Build a fragment of R code with the determinants:
16 code = sprintf("max(%s)", string_from_floats(dets));
17 r = R(code);
18 printf("dets: max: %f", r);
```

Figure 12. Swift script for algorithm combining NumPy and R.

This method could be extended to call to C, C++, or Fortran numerical libraries as well. The various Swift data may be used, including raw binary data. Note that in practice, one normally would call to application components, not numerical libraries, but this example illustrates the generality of our approach.

One current deficiency in this technique relative to the direct use of NumPy or R is that Swift does not provide the convenient mathematical syntax available in NumPy or R (for example, in NumPy, one may multiply matrices A and B with `A*B` using the provided overloaded operator). Future work will address this deficiency.

V. Performance

Swift/T performance has been reported elsewhere [2], [13]. In this work, we report on the capability of Swift/T to rapidly launch many Python and R tasks.

This script simply performs a parallel Swift `foreach` loop around a call into Python to render the result of `2+2` as a string. The string is returned to Swift/T (the call to Python is not optimized out as an unused value because of the potential for side effects). Each call to `python()` instantiates a fresh Python instance to capture the full cost of using Python from Swift.

```
1  main {
2    N = toint(argv("N"));
3    printf("N: %i\n", N);
4    foreach i in [0:N-1] {
5      python("'{0}'.format(2+2)");
6    }
7  }
```

Figure 13. Swift script used for Python task rate measurements.

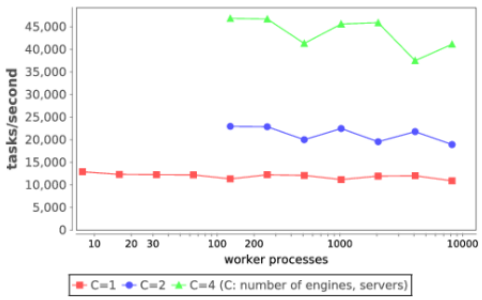

Figure 14. Rates for Python tasks on *Vesta* - varying worker processes.

The Python execution occurs only on Swift/T worker processes. This means that each task is produced by `foreach` evaluation on an engine, load balanced by an ADLB server, and executed on a worker.

In our first tests, we used *Vesta*, a 2,048-node, 32,768-core Blue Gene/Q at the Argonne Leadership Computing Facility (ALCF). Each node contains 16 PowerPC A2 cores running at 1.6 GHz and 16 GB RAM connected to a low-latency 5D torus interconnect. The Swift script used is shown in Figure 13.

In this test, we measure the ability of Swift/T to rapidly launch Python interpreters for an increasing number of workers up to 8,192 (on the x axis) and a varying number of control processes: for `C=1`, there is one engine and one server, and so on.

Results are shown in Figure 14. This shows that for each number of control processes C, the performance is the same for any number of workers. This indicates that performance is limited by the control processes and not by launching Python.

In the next test, we fix the number of workers W and increase C. For $W = 4{,}096$, performance scales linearly up to 64 engines and servers, after which performance degrades. For $W = 8{,}192$, up to 128 engines and servers may be used.

In the final test, we measure the ability of Swift/T to rapidly launch Python interpreters on many processors of *Blue Waters*, a combination Cray XE6 and XK7 system. In the XE component, each of the 22,640 nodes contains 16 AMD Interlagos cores running at 2.3 GHz and 64 GB

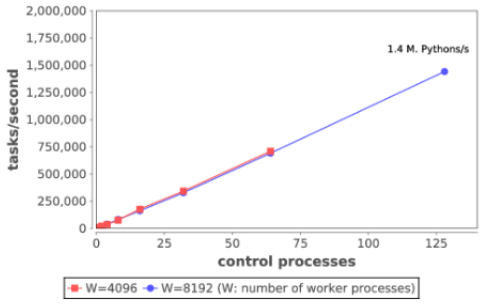

Figure 15. Rates for Python tasks on *Vesta* - varying control processes.

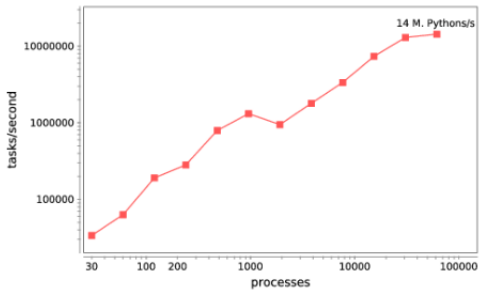

Figure 16. Rates for Python tasks on Blue Waters - varying total processes.

RAM connected by a low latency 3D torus interconnect. We performed this loop for an increasing number of processors up to 65,536 (on the x axis) and a varying number of control processes: for each number of total processes P, the number of control processes C is configured such that $C = P/32$.

The results in Figure 16 show that, using Swift/T, our script was able to utilize 65,536 cores well, instantiating approximately 14 million Python interpreters per second. *Blue Waters* contains only 33,792 XK cores in nodes with GPUs, so this demonstrates that the model proposed for the OOPS application in § II-B is viable.

VI. Related Work

In this section, we explore related work from two broad areas: 1) Python implementations for scientific, parallel and distributed computing, and 2) implementations addressing language interoperability challenges.

With an ever growing popularity of Python and support of an active developer community, the language has enjoyed a significant following in the scientific computing community. Packages such as NumPy, SciPy, and matplotlib offer useful numeric, scientific, and visualization utilities. These packages became a strong foundation for full-fledged Python-based platforms for scientific computing such as IPython Parallel [14]. The IPython system is a Python package that evolved from an alternative interactive Python terminal (from the SciPy community) into a message-based parallel and distributed computing platform. IPython provides many features suitable for scientific computing such as interactive visualization and the scientific library SciPy.

The Celery [15] project provides parallel programming methods for multi- and many-core node architectures. Celery, based in Python, offers an implementation of task-queue with tools that provide mechanisms to define workflows, monitoring, and cron-like task scheduling. Celery can use a third-party messaging library such as RabbitMQ or MongoDB for inter-task and task-client communications.

Language interoperability is an invaluable capability for legacy codes because it allows existing code to be reused for new, advanced systems without tedious and error-prone code rewrites. Many tools for language interoperability exist. SWIG [16] is a tool that offers the ability to interface code written in low-level languages with high-level scripting languages. It allows language-level invocations such as function calls to be exposed as external callable functions. Similarly, the Java Native Interface (JNI) [17] offers a Java API to interface with C/C++ code. Swift/T uses SWIG directly and uses the same concepts to bring lower-level code into the Swift/T framework.

Babel [18] is a high performance language interoperability tool based on the Scientific Interface Description Language (SIDL). This allows transmission of data types from one language to another. Swift/T differs from Babel/SIDL in that it started as a scalable dataflow framework that is now adding interlanguage features. We will investigate the compatibility of Babel/SIDL concepts with Swift/T in future work.

VII. Conclusion

One way to solve a problem is to make it bigger... [19]

Modern scientific application development is trending toward greater software complexity and more demanding performance requirements. These applications blend structured and unstructured computing patterns, features for distributed and parallel computing, and the use of specialized libraries for everything from numerics to I/O. For continued progress in scientific computing, tools must be developed and adopted that enable rapid prototyping and development of complex, large scale applications.

In this work, we provided a broad overview of relevant scientific computing applications that combine computing patterns and use multiple languages. We described the Swift/T system for high-performance computing, highlighted its new features to support scripting languages Python and R, and showed how these can be combined to solve numerical problems. We then provided performance results from large-scale synthetic runs using these technologies.

In future work, we intend to improve support for external languages by improving support for more complex data types. Additionally, we will investigate syntactic features in Swift/T

to provide more appropriate syntax for numerical expressions beyond our current functional syntax. Future applications are sure to challenge the current performance envelope, and we will improve and apply our techniques to solve bigger problems with more advanced tools on the largest scale machines.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We thank Ray Osborn for collaboration on NeXus, Victor Zavala for collaboration on power grid, and Reinhard Neder for collaboration on DISCUS.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

REFERENCES

[1] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, pp. 633–652, 2011.

[2] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Scalable data flow programming for many-task applications," in *Proc. CCGrid*, 2013.

[3] P. Klosowski, M. Koennecke, J. Tischler, and R. Osborn, "NeXus: A common format for the exchange of neutron and synchrotron data," *Physica B: Condensed Matter*, vol. 241243, pp. 151 – 153, 1997.

[4] M. Folk, R. McGrath, and N. Yeager, "HDF: An update and future directions," in *Proc. Geoscience and Remote Sensing Symposium*, 1999.

[5] R. Osborn, "NeXpy web site," http://nexpy.github.io/nexpy.

[6] S. van der Walt, S. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Computing in Science Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[7] A. N. Adhikari, J. Peng, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick, "Modeling large regions in proteins: Applications to loops, termini, and folding," *Protein Science*, vol. 21, no. 1, pp. 107–121, 2012.

[8] S. J. Krieder and I. Raicu, "Towards the support for many-task computing on many-core computing platforms," Doctoral Showcase, IEEE/ACM Supercomputing/SC, 2012.

[9] T. Proffen and R. Neder, "DISCUS: A program for diffuse scattering and defect-structure simulation," *Journal of Applied Crystallography*, vol. 30, no. 2, pp. 171–175, 1997.

[10] K. Price, R. Storn, and J. Lampinen, *Differential evolution*. Springer, 2005.

[11] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, pp. 30–37, January 2010.

[12] J. M. Wozniak, T. G. Armstrong, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed memory data flow engine for many-task applications," in *Int'l Workshop Scalable Workflow Enactment Engines and Technologies (SWEET) 2012*, 2012.

[13] J. M. Wozniak, T. Peterka, T. G. Armstrong, J. Dinan, E. Lusk, M. Wilde, and I. Foster, "Dataflow coordination of data-parallel tasks via MPI 3.0," in *Proc. EuroMPI*, 2013.

[14] F. Pérez and B. E. Granger, "IPython: A system for interactive scientific computing," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 21–29, May 2007.

[15] "Celery: Distributed Task Queue," celeryproject.org.

[16] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proc. USENIX Tcl/Tk Workshop*, 1996.

[17] R. Gordon, *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., 1998.

[18] "BABEL: High-performance language interoperability," http://computation.llnl.gov/casc/components.

[19] SyFy, "TV trailer for Mega Python vs. Gatoroid," 2011.