Interlanguage Parallel Scripting for Distributed-Memory Scientific Computing

Justin M. Wozniak
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL USA
wozniak@mcs.anl.gov

Timothy G. Armstrong
Computer Science Department
University of Chicago
Chicago, IL USA
tga@uchicago.edu

Ketan C. Maheshwari
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL USA
ketan@mcs.anl.gov

Daniel S. Katz
Computation Institute
University of Chicago &
Argonne National Laboratory
Chicago, IL USA
d.katz@ieee.org

Michael Wilde
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL USA
wilde@mcs.anl.gov

Ian T. Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL USA
foster@mcs.anl.gov

ABSTRACT

Scripting languages such as Python and R have been widely adopted as tools for the development of scientific software because of the expressiveness of the languages and their available libraries. However, deploying scripted applications on large-scale parallel computer systems such as the IBM Blue Gene/Q or Cray XE6 is a challenge because of issues including operating system limitations, interoperability challenges, and parallel filesystem overheads due to the small file system accesses common in scripted approaches. We present a new approach to these problems in which the Swift scripting system is used to integrate high-level scripts written in Python, R, and Tcl with native code developed in C, C++, and Fortran, by linking Swift to the library interfaces to the script interpreters. We present a technique to efficiently launch scripted applications on supercomputers, and we demonstrate high performance, such as invoking 14M Python interpreters per second on *Blue Waters*.

1. INTRODUCTION

An increasing number of 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 (C, C++, Fortran) with high-level functionality expressed in rapidly developed and modified scripts. Additional specialized language-specific features may be used for concurrency, I/O, the use of accelerators, and so on. These development techniques have been used in a wide range of application domains, from materials science and protein analysis to power grid simulation.

SC '15, Austin, Texas USA

DOI: 10.1145/2822332.2822338

Such applications and tools are commonly developed as follows. First, a native code library is built or repurposed for the core processing. Second, a collection of scripts is built up around the core library or program to express the complex, often dynamic, but less performance-critical coordination logic. Such "wrapper scripts" may be developed with shell, Python, Tcl, or other tools. Third, when additional scalability is required, native code or additional scripts are developed to deploy the application in some distributed computing model such as MPI, Swift, some other grid workflow system or with custom wrapper scripts that submit jobs to a scheduler such as PBS.

Swift [23] is a programming language and runtime designed to ease this software development methodology. 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 [26], generates an MPI program from the Swift script and provides tools to run that program on various scheduled resources. This approach has allowed Swift/T to scale the execution of scripted applications to hundreds of thousands of cores [2]. In this approach, Swift handles data management, movement, and marshaling among distributed-memory processes without direct user manipulation of low-level communication libraries such as MPI.

The Swift/T framework supports direct calls to native code through library loading and access. Modern scientific applications, however, are built not only with native code but also 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 that support direct calls to Python, R, and Tcl. These features, which could easily be extended to other scripting languages, 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.

Indeed, *any* external program 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 (say, in Python). 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 laborious manner. The developer would have to build significant infrastructure to 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, that approach 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. Swift data is passed among language-specific components implicitly as the script progresses; no user data marshaling is required. (MPI messaging is used internally by the Swift/T runtime.) 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 rest of this paper is organized as follows. In Section 2 we describe relevant application models. In Section 3 we describe the architecture of Swift/T, and in Section 4 we describe the interlanguage features that are the focus of this paper. In Section 5 we present performance results from Swift/T running on Blue Gene and Cray systems. In Section 6, we describe related work and in Section 7 we offer concluding remarks and a glimpse of future work.

2. APPLICATIONS

In this section, we describe some interlanguage applications that have motivated this work. We use them to define a general application model.

2.1 NeXus: Storage and processing for materials science

Our first motivating application concerns the distributed storage and analysis of large materials science datasets. These datasets are too large to move in their entirety for processing and visualization; instead, we want to be able to extract, transfer, and manipulate selected subsets. Interlanguage support allows us to combine Swift for distribution with Python for analysis and visualization.

NeXus [11] is an HDF-based [8] file format for X-ray, neutron, and muon scattering data. By standardizing discipline-specific metadata stored in the HDF file, NeXus makes X-ray scattering data easier to distribute and analyze by different research groups. A typical NeXus dataset may include multiple large (up to 30 GB) multidimensional (up to 4D) arrays. Datasets are typically processed repeatedly for coordinate transformation, analysis, and visualization. Fragments of the underlying data variables may be processed concurrently in a task-oriented model.

NeXpy [16] is a Python-based GUI and scripting suite for operating on NeXus data. Many important data transformations, analysis routines, and visualization capabilities are made available through its scripted Python API. NeXpy users can, for example, easily rotate NeXus scattering data and perform common visualization operations. Underlying numerical operations are performed with NumPy [22], a numerical library for Python, as shown in Figure 1.

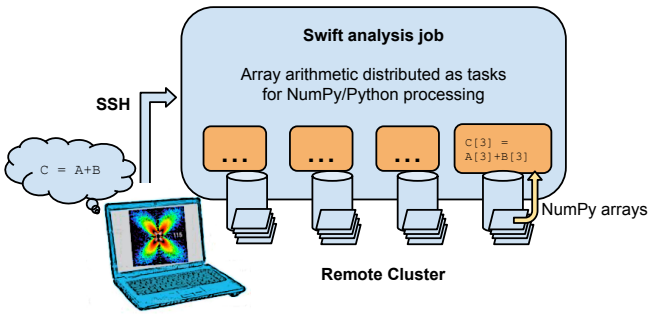

Figure 1: Parallel data analysis via Swift/NumPy processing in cluster.

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. To reuse as much existing NeXpy code as possible, we want to be able to call relevant functionality as Python tasks from Swift. To do so effectively, we need interlanguage support to pass NeXus/NumPy data from Swift to Python and back.

2.2 OOPS: Protein simulation

The development of molecular dynamics simulation ensembles can benefit greatly from the application of dynamic languages [18]. While the core force computations and timestepping must be performed with native code, experiment-level control is best performed in a high-level script.

The Open Protein Simulator (OOPS) is an interlanguage implementation of a protein folding simulation code [1]. 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. The modular OOPS architecture 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 [12]. In this effort, the OOPS libraries are being 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.

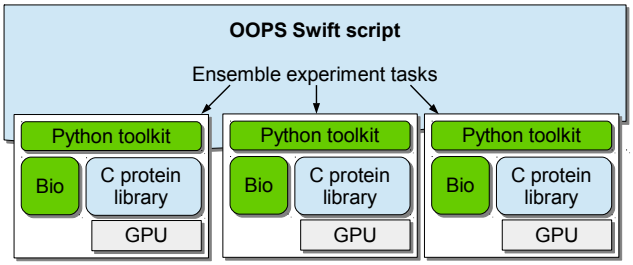

Figure 2: Component architecture of OOPS application.

2.3 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 software to carry out these studies combines performance-critical mathematical programming (AMPL [9]) with a higher level branch-and-bound framework. While AMPL is a native code library, many application-specific scripts written in Python generate sample scenarios (weather, etc.). Thus, a distributed-memory tool to conveniently manage these codes and organize the overall application is desirable.

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:

```
1  int nS[] = [10, 100, 1000, 10000, 100000];
2  foreach S, idxs in nS {
3    sample0 = gensample(wind_data);
4    obj[idxs] = ampl(sample0);
5    foreach B, idxb in [10:40:10] {
6      foreach k in [0:B]{
7        sample1 = gensample(S, wind_data);
8        obj_l[idxs][idxb][k] = ampl_L(sample1);
9        sample2 = gensample(S, wind_data);
10       obj_u[idxs][idxb][k] = ampl_U(sample2,
11                                     obj[idxs]);
12 }}}
```

In this code, `gensample()` is not a pure function— 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 application calls. Each application call uses the Python-implemented sample generator (`gensample()`) and the AMPL models, making it an interlanguage implementation spanning Python and AMPL interpreters, as depicted in Figure 3.

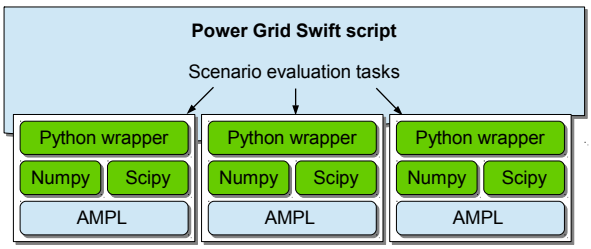

Figure 3: Electrical power price analysis application components.

2.4 DISCUS: Crystal structure scattering simulation

DISCUS [21] 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 with results of a real experiment, allowing the accuracy of the input parameters to be gauged. An evolutionary algorithm is used [20] in order to iteratively adjust the parameters to improve the fit, as shown in Figure 4.

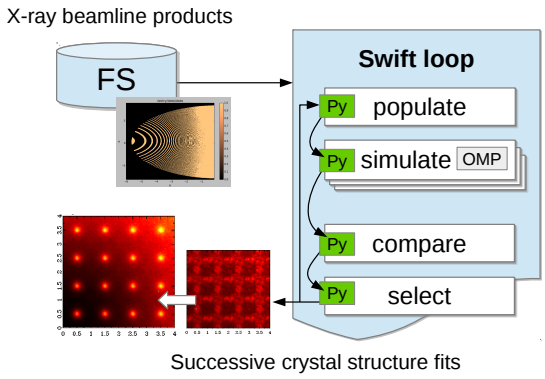

Figure 4: Multilevel parallelism in DISCUS.

Two levels of parallelism have been identified in this compute-intensive process. First, each individual DISCUS run can be improved through the application of thread parallelism in OpenMP. Second, the 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.

In the Swift model, we applied interlanguage processing. Features from the DISCUS package were wrapped with Python via `f2py`. Then the application was rapidly constructed as

shown.

Computationally, this algorithm is an exciting use case for the application of HPC for diffuse scattering. While we have run up to 512 concurrent simulations in the evolutionary population (see §5.2), the method can make use of at least 5000 concurrent simulations. Furthermore, we have incorporated OpenMP support into a performance-critical DISCUS method, allowing each simulation to use ∼20 threads. Thus, we can use ∼100,000 cores concurrently on a large-scale supercomputer.

2.5 Generic application models

We generalize from these four real-world applications to define a general model for interlanguage scientific applications.

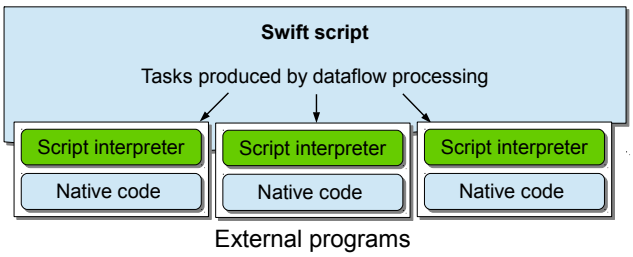

Figure 5: Existing software model previously supported by Swift.

In the model shown in Figure 5, existing application components of native code libraries wrapped in high-level languages are then expressed by Swift. This approach allows the reuse of application logic while providing concurrency at the task level.

A subtle change is introduced in the model shown in Figure 6. *In this model, high-level language components are brought conceptually close to the Swift level as a result of tight interlanguage support.* It also offers a performance boost due to linking to the language library (as opposed to calling the interpreter as an external program).

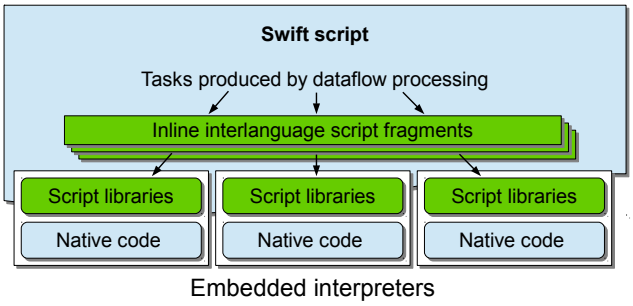

Figure 6: New software model supported by present work. High-level languages (Python, R, etc.) are integrated closely with the Swift script but access existing script or native code components.

3. ARCHITECTURE

We next provide some background on the Swift language, describe the Swift/T architecture, and discuss how Swift/T calls application components.

3.1 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(i)` via `t`. The code implies the dataflow dependencies shown in Figure 7, where several parallel pipelines of tasks are present. Swift will construct and execute these pipelines in parallel on any available resources.

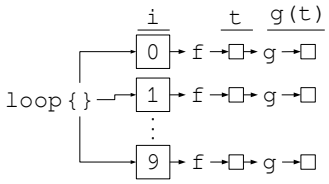

Figure 7: Diagram of implicit dataflow of Swift loop.

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 runtimes, the asynchronous, load-balanced Swift model is an excellent fit.

3.2 Swift/T runtime

Swift/T [26] is a reimplementation of the Swift/K [23] framework for high-performance computing.

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.

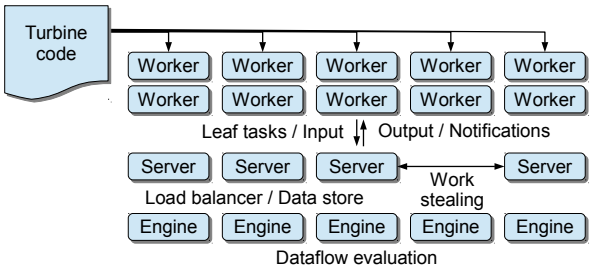

Figure 8: Swift/T runtime architecture.

Swift/T is designed for high-performance computing at the largest scale. T indicates that the key features are implemented by the Turbine dataflow engine [24]. In this implementation of Swift, the Swift script is translated into a runtime framework based on the MPI-based Asynchronous Dynamic Load Balancer (ADLB) [14] and Turbine libraries, which evaluate Swift semantics in a distributed manner (no bottleneck). Thus, at run time, Swift/T programs are MPI programs.

The Swift/T architecture is diagrammed in Figure 8. Each MPI 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.

4. INTERLANGUAGE OPERATIONS

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

4.1 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 by using a 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 the 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 `argc`/`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-related errors common in shell scripting are easily avoided.

4.2 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) an interpreter that does not require the user to run the C compiler in order to avoid complexities on advanced systems. Thus, we chose Tcl to represent Turbine code, and we made use of the ease of calling C from Tcl in order 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. The 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 be 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 $<< \cdot >>$ 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 allows for the rapid development of Swift builtin functions 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 by using Swift support for Tcl packages. Fourth, the strength of Tcl support for calling native code is easily brought into Swift as well, as described in the following subsection.

4.3 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 [3] 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 [27].

In order to call into an existing native code program from Swift, the following steps are necessary. 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 [15], 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 the Swift/T code.

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 it 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 approach 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,

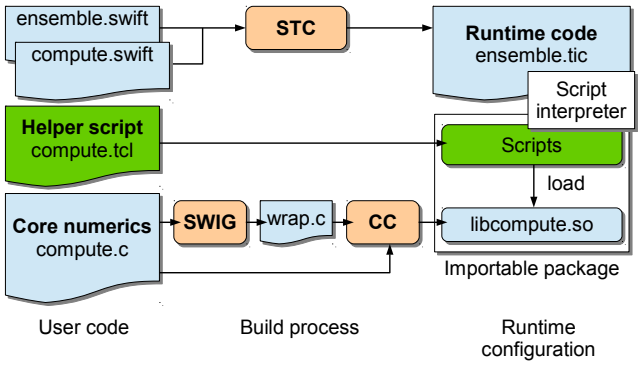

Figure 9: SWIG providing Tcl bindings for C functions callable from Swift/T.

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*` – instead, `blobutils` provides tools to handle the simple but myriad interlanguage complexities found when operating on binary data.

4.4 Calling Python or R

As described above, many modern scientific applications have key components or interfaces built in Python, R, or other high-level languages. Previous workflow programming systems typically call external languages by executing the external interpreter executables. This strategy is undesirable for Swift/T, however, 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.

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 can 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. Users interact only with the high-level Swift/T leaf function, greatly reducing complexity.

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 this state is inside the Swift/T MPI run). One approach is to finalize the interpreter at the end of each task and reinitialize it when the next task is started, thus clearing any state. This approach raises concerns about performance and possible resource leaks. Thus, we provide options to either retain the interpreter or reinitialize it. (Old interpreter state can also be used to store useful data if the programmer is careful.)

4.5 Calling Python and R

In addition to supporting rapid, concurrent calls to scripted application components, Swift/T 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.

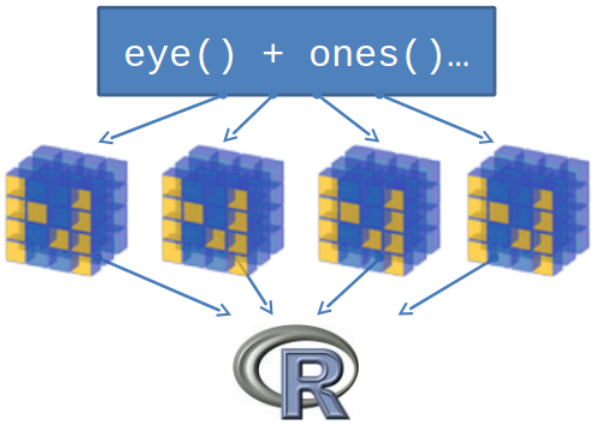

Figure 10: Graphical depiction of algorithm combining NumPy and R.

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 $A = I_N * i + 1$, and 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`. Fourth, the R function `max()` is used to obtain the maximal value.

This method could be extended to call to C, C++, or Fortran numerical libraries as well. The various Swift data may

```
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.

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.

4.6 Calling Julia

Julia [4] is a mathematically oriented scripting language notable for its high performance, ease of calling from Julia to C, and natural parallel capabilities. One can also call from C to Julia. Julia is thus an important target for Swift/T, which can be used to organize large numbers of Julia runs in a composite, interlanguage application.

Julia provides a C header file and set of functions that may be used to initialize a Julia interpreter and evaluate string commands in that interpreter. A use of Julia from Swift is shown in Figure 13, which simply increments the given integer after a 1-second delay.

```
1  f =
2  """
3  begin
4   f(x) = begin
5            sleep(1)
6            x+1
7          end
8   f(%s)
9  end
10 """;
11 s1 = julia(sprintf(f, 1));
```

Figure 13: Swift script for use of Julia.

4.7 Addressing filesystem contention with static packages

A significant concern when using a scripted approach on large supercomputers is the impact on the parallel file system. Although such file systems are optimized for large data transfers, they are easily overwhelmed by large numbers of small file operations issued by many nodes simultaneously.

In a typical scripted approach, many system and user script files are dynamically located and interpreted at runtime, in addition to myriad interpreter and user libraries. Small application configuration files may also be read, possibly for each task in the workflow.

Supercomputers such the Cray XE6 and Blue Gene/Q use specialized techniques to broadcast the executable to compute nodes before execution, but this feature is not exposed to the user. Our approach is to make use of this feature by bundling everything into a static executable, piggybacking on the broadcast feature, unpacking into node-local storage (e.g., RAM drive), and running from there. We can pack everything into this executable, including user scripts and small data files.

The user follows this procedure at build time:

1. Fill in a *Swift/T manifest file* that contains any desired scripts, libraries, or data files to be placed in the executable.

2. Run the provided `mkstatic` script that generates a single C file, containing a `main()` function. This file includes user scripts and all Swift/T supporting scripts (about 24 Tcl files) and is encoded for the C compiler by `xxd -i`.

3. Compile the generated C file into a generated object file.

4. Link the object file with a provided list of Swift/T runtime libraries and any user libraries. In some cases, this produces a pure static executable. On some systems, some static libraries are not available, but the linker options are available to link as many static libraries as possible.

Then, at run time, the following occurs:

1. The executable is distributed to all compute nodes by the efficient broadcast mechanism.

2. If the executable was not a pure static executable, any dynamic libraries are loaded by the OS.

3. Swift/T runtime scripts are interpreted from the strings encoded in the generated C source code.

4. User scripts and data are unpacked from the encoded data into local storage.

5. The Swift/T program begins. User tasks may access local storage to interpret scripts and configuration files, or load small data.

Following this approach, an ad hoc, scripted approach is able to execute efficiently on a large-scale machine.

5. PERFORMANCE

Swift/T performance has been reported elsewhere [26, 27]. In this work, we report on the capability of Swift/T to rapidly launch many Python and R tasks.

```
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 14: Swift script used for Python task rate measurements.

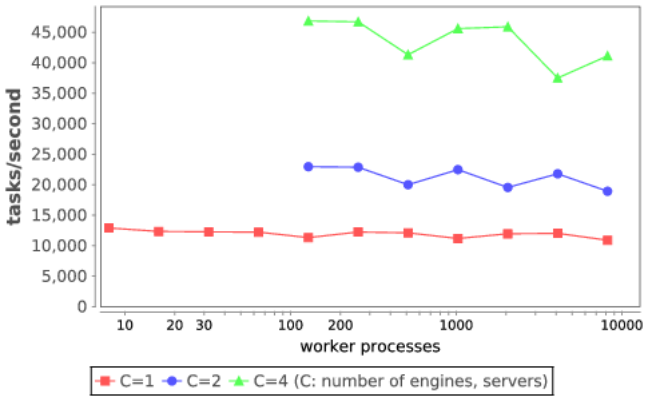

Figure 15: Rates for Python tasks on *Vesta* - varying worker processes.

5.1 Plain Python

The script in Figure 14 simply performs a parallel Swift `foreach` loop around a call into Python to render the result of `2+2` as a string. Each call to `python()` instantiates a fresh Python instance to capture the full cost of using Python from Swift.

The Python execution occurs only on Swift/T worker processes. In other words, 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.

In this test, we measured the ability of Swift/T to rapidly launch Python interpreters for an increasing number of workers up to 8,192 and a varying number of control process pairs: for `C=1`, there is one engine and one server, and so on (see § 3.2).

Results are shown in Figure 15. We can see that for each control number C, the performance is the same for any number of workers (on the x axis). This indicates that performance is limited by the control processes and not by launching Python.

In the next test, shown in Figure 16, we fixed the number of workers W and increased 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 measured the ability of Swift/T to rapidly launch Python interpreters on many processors of *Blue Waters*, a combination Cray XE6 and XK7 system.

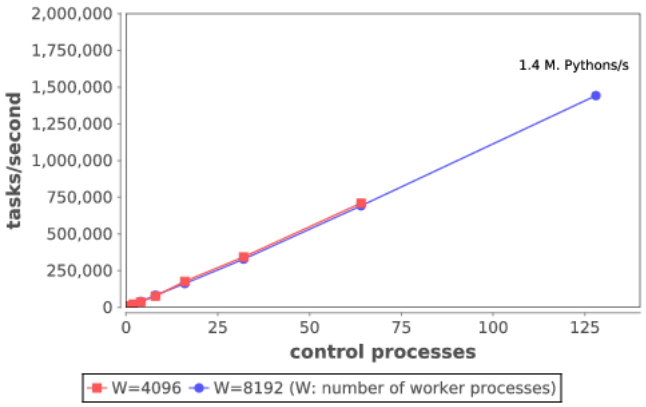

Figure 16: Rates for Python tasks on *Vesta* - varying control processes.

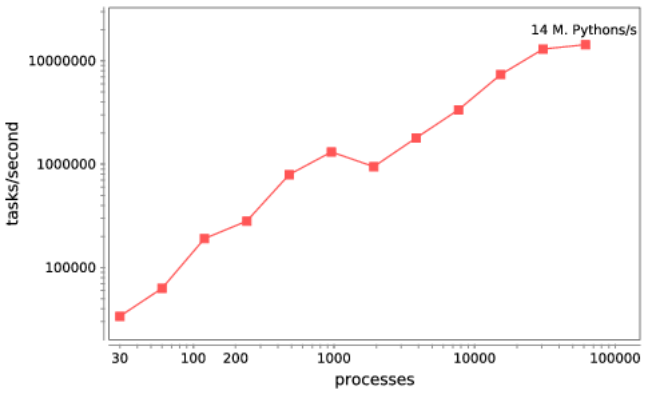

Figure 17: Rates for Python tasks on *Blue Waters* - varying total processes.

In the XE component, each of the 22,640 nodes contains 16 AMD Interlagos cores running at 2.3 GHz and 64 GB 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 17 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 § 2.2 is viable.

5.2 DISCUS evolutionary algorithm

The DISCUS application was described in §2.4. We ran this application on *Beagle*, a Cray XE6 at the University of Chicago. The application was run without OpenMP enabled (since OpenMP parallelism does not stress Swift task distribution), and ran it on up to 512 processes. Tasks in the run perform computation for varying lengths of time. As shown in Figure 18, the application scales as expected.

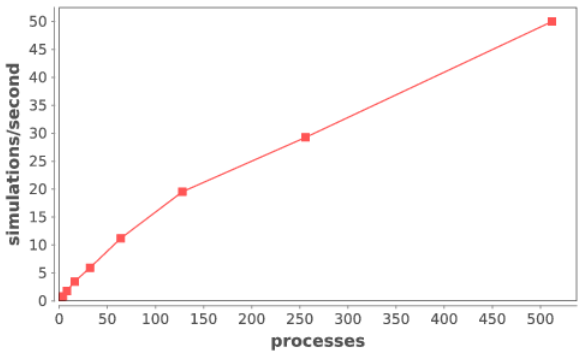

Figure 18: Rates for DISCUS tasks on Beagle - varying total processes.

6. RELATED WORK

Along with ever-growing popularity and support of an active developer community, the Python language enjoys 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 Python-based platforms for scientific computing such as IPython Parallel [17]. 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.

The Celery [5, 13] 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 [3] 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) [10] offers a Java API to interface with C/C++ code. Swift/T uses the same concepts to bring lower-level code into the Swift/T framework.

Babel [7] is a high performance language interoperability tool based on the Scientific Interface Description Language (SIDL), allowing transmission of typed data among languages. Swift/T differs from Babel/SIDL in that it started as a scalable dataflow framework that is now adding inter-language features. We will investigate the compatibility of Babel/SIDL concepts with Swift/T in future work. Babel was used to extend Chapel [6] to call traditional programming languages (but did not investigate scripting languages such as Python in detail) [19].

7. CONCLUSION

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. This paper extended a previous short paper [25] with additional language coverage and all performance results.

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 such as Python and R, and showed how these can be combined to solve numerical problems. We described our use of embedded script interpreters, making interlanguage programming relatively easy while remaining compatible with systems having restricted OS functionality such as the IBM Blue Gene/Q. Additionally, we addressed the many small file problem common in scripted solutions with our support for creating static executables.

We then provided performance results from large-scale synthetic runs using these technologies, running on two Cray systems and an Blue Gene/Q . We also addressed the problem of many small file accesses typical of scripted approaches, and described our solution in detail. In addition, we presented a scaling result for a real-world X-ray science application.

In future work, we intend to improve support for external languages by improving support for automatically translating more complex data types. 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.

Swift/T is available at `http://swift-lang.org/Swift-T`.

Acknowledgments

This material was based upon work supported 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.

Work by Katz was supported by the National Science Foundation while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

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.

8. REFERENCES

[1] 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*, 21(1):107–121, 2012.

[2] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC*, 2014.

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

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing, 2014. http://arxiv.org/abs/1411.1607.

[5] Celery: Distributed Task Queue. celeryproject.org.

[6] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[7] T. W. Epperly, G. Kumfert, T. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn. High-performance language interoperability for scientific computing through Babel. *J. of High-performance Computing Applications*, 26(3), 2012.

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

[9] R. Fourer, D. M. Gay, and B. Kernighan. AMPL: a mathematical programming language. In S. W. Wallace, editor, *Algorithms and model formulations in mathematical programming*, pages 150–151. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

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

[11] 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*, 241âĂŞ243:151 – 153, 1997.

[12] 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.

[13] M. Lunacek, J. Braden, and T. Hauser. The scaling of many-task computing approaches in Python on cluster supercomputers. In *Proc. CLUSTER*, 2013.

[14] 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*, 17:30–37, January 2010.

[15] J. McFarland. FortWrap web site. http://fortwrap.sourceforge.net.

[16] R. Osborn. NeXpy web site. http://nexpy.github.io/nexpy.

[17] F. Pérez and B. E. Granger. IPython: A system for interactive scientific computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[18] J. C. Phillips, J. E. Stone, K. L. Vandivort, T. G. Armstrong, J. M. Wozniak, M. Wilde, and K. Schulten. Petascale Tcl with NAMD, VMD, and Swift/T. In *Proc. High Performance Technical Computing in Dynamic Languages at SC*, 2014.

[19] A. Prantl, T. Epperly, S. Imam, and V. Sarkar. Interfacing Chapel with traditional HPC programming languages. In *Proc. Partitioned Global Address Space Programming Models*, 2011.

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

[21] T. Proffen and R. Neder. DISCUS: A program for diffuse scattering and defect-structure simulation. *Journal of Applied Crystallography*, 30(2):171–175, 1997.

[22] S. van der Walt, S. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, 2011.

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

[24] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A distributed-memory dataflow engine for high performance many-task applications. *Fundamenta Informaticae*, 28(3), 2013.

[25] J. M. Wozniak, T. G. Armstrong, K. C. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster. Toward interlanguage parallel scripting for distributed-memory scientific computing. In *Proc. CLUSTER*, 2015.

[26] 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.

[27] 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.