Swift: A language for distributed parallel scripting

Michael Wilde[a,b,*], Mihael Hategan[a], Justin M. Wozniak[b], Ben Clifford[d], Daniel S. Katz[a], Ian Foster[a,b,c]

[a] *Computation Institute, University of Chicago and Argonne National Laboratory*
[b] *Mathematics and Computer Science Division, Argonne National Laboratory*
[c] *Department of Computer Science, University of Chicago*
[d] *Department of Astronomy and Astrophysics, University of Chicago*

Abstract

Scientists, engineers, and statisticians must execute domain-specific application programs many times on large collections of file-based data. This activity requires complex orchestration and data management as data is passed to, from, and among application invocations. Distributed and parallel computing resources can accelerate such processing, but their use further increases programming complexity. The Swift parallel scripting language reduces these complexities by making file system structures accessible via language constructs and by allowing ordinary application programs to be composed into powerful parallel scripts that can efficiently utilize parallel and distributed resources. We present Swift's implicitly parallel and deterministic programming model, which applies external applications to file collections using a functional style that abstracts and simplifies distributed parallel execution.

Keywords: Swift, parallel programming, scripting, dataflow

1. Introduction

Swift is a scripting language designed for composing application programs into parallel applications that can be executed on multicore processors, clusters, grids, clouds, and supercomputers. Unlike most other scripting languages, Swift focuses on the issues that arise from the concurrent

*Corresponding author
Email address: wilde@mcs.anl.gov (Michael Wilde)

execution, composition, and coordination of many independent (and, typically, distributed) computational tasks. Swift scripts express the execution of programs that consume and produce file-resident datasets. Swift uses a C-like syntax consisting of function definitions and expressions, with dataflow-driven semantics and implicit parallelism.

Many parallel applications involve a single message-passing parallel program: a model supported well by the Message Passing Interface (MPI). Others, however, require the coupling or orchestration of large numbers of application invocations: either many invocations of the same program or many invocations of sequences and patterns of several programs. Scaling up requires the distribution of such workloads among cores, processors, computers, or clusters and, hence, the use of parallel or grid computing. Even if a single large parallel cluster suffices, users will not always have access to the same system (i.e., big machines may be congested or temporarily unavailable to a user because of maintenance or allocation depletion). Thus, it is desirable to be able to use whatever resources happen to be available or economical at the moment when the user needs to compute—without the need to continually reprogram or adjust execution scripts.

Swift's primary value is that it provides a simple, minimal set of language constructs to specify how applications are glued together and executed in parallel at large scale. It regularizes and abstracts notions of processes and external data for distributed parallel execution of application programs.

Swift is implicitly parallel and location-independent: the user does not explicitly code either parallel behavior or synchronization (or mutual exclusion) and does not code explicit transfer of files to and from execution sites. In fact, no knowledge of runtime execution locations is directly specified in a Swift script. The function model on which Swift is based ensures that execution of Swift scripts is deterministic (if the called functions are themselves deterministic), thus simplifying the scripting process. Having the results of a Swift script be independent of the way that its function invocations are parallelized implies that the functions must, for the same input, produce the same output, irrespective of the time, order, or location in which they are executed. However, Swift greatly simplifies the parallel scripting process even when this condition is not met.

As a language, Swift is simpler than most scripting languages because it does not replicate the capabilities that scripting languages such as Perl, Python, and shells do well; instead, Swift makes it easy to call such scripts as small applications.

Swift can execute scripts that perform hundreds of thousands of program invocations on highly parallel resources and handle the unreliable and dynamic aspects of wide-area distributed resources. Such issues are managed by Swift's runtime system and are not manifest in the user's scripts. The exact number of processing units available on such shared resources varies with time. In order to take advantage of as many processing units as possible during the execution of a Swift program, flexibility is essential in the way the execution of individual processes is parallelized. Swift exploits the maximal concurrency permitted by data dependencies within a script and by external resource availability.

Swift enables users to specify process composition by representing processes as functions, where input data files and process parameters become function parameters and output data files become function return values. Swift also provides a high-level representation of collections of data (used as function inputs and outputs) and a specification ("mapper") that allows those collections to be processed by external programs. We chose to make the Swift language purely functional (i.e., all operations have a well-defined set of inputs and outputs, all variables are write-once, and no script-level side effects are permitted by the language) in order to prevent the difficulties that arise from having to track side effects to ensure determinism in complex concurrency scenarios. Functional programming allows consistent implementations of evaluation strategies, in contrast to the more common approach of eager evaluation. This benefit has been similarly demonstrated in lazily evaluated languages such as Haskell [1].

In order to achieve automatic parallelization, Swift is based on the synchronization construct of *futures* [2], which can enable large-scale parallelism. Every Swift variable (including all members of structures and arrays) is a future. Using a futures-based evaluation strategy allows for automatic parallelization without the need for dependency analysis. This significantly simplifies the Swift implementation.

We believe that the missing feature in current scripting languages is sufficient specification and encapsulation of inputs to and outputs from a given application, such that an execution environment can automatically make remote execution transparent. Without this, achieving location transparency is not feasible. Swift adds to scripting what the remote procedure call (RPC) paradigm [3] adds to programming: by formalizing the inputs and outputs of applications that have been declared as Swift functions, it makes the distributed remote execution of applications transparent.

Swift has been described previously [4]; this paper goes into greater depth in describing the parallel aspects of the language, the way its implementation handles large-scale and distributed execution environments, and its contribution to distributed and parallel programming models.

The remainder of this paper is organized as follows. Section 2 presents the fundamental concepts and language structures of Swift. Section 3 details the Swift implementation, including the distributed architecture that enables applications to run on distributed resources. Section 4 describes real-world applications using Swift on scientific projects. Section 5 provides performance results. Section 6 relates Swift to other systems. Section 7 highlights ongoing and future work in the Swift project. Section 8 offers concluding remarks about Swift's distinguishing features and its role in scientific computing.

2. The Swift language

Swift is, by design, a sparse scripting language that executes external programs remotely and in parallel. As such, Swift has only a limited set of data types, operators, and built-in functions. Its simple, uniform data model comprises a few atomic types (that can be scalar values or references to external files) and two collection types (arrays and structures).

A Swift script uses a C-like syntax to describe data, application components, invocations of application components, and the interrelations (data flow) among those invocations. Swift scripts are written as a set of functions, composed upwards, starting with *atomic functions* that specify the execution of external programs. Higher-level functions are then composed as pipelines (or, more generally, graphs) of subfunctions.

Unlike most other scripting languages, Swift expresses invocations of ordinary programs—technically, POSIX `exec()` operations—in a manner that explicitly declares the files and command-line arguments that are the inputs of each program invocation. Swift scripts similarly declare all output files that result from program invocations. This approach enables Swift to provide distributed, location-independent execution of external application programs.

The Swift parallel execution model is based on two concepts that are applied uniformly throughout the language. First, every Swift data element behaves like a *future*. By "data element" we mean both the named variables within a function's environment, such as its local variables, parameters, and

returns, and the individual elements of array and structure collections. Second, all expressions in a Swift program are conceptually executed in parallel. Expressions (including function evaluations) wait for input values when they are required and then set their result values as their computation proceeds. These fundamental concepts of pervasive implicit parallelism and transparent location independence, along with natural manner in which Swift expresses the processing of files by applications as if they were "in-memory" objects, are the powerful aspects that make Swift unique among scripting tools. These aspects are elaborated in this section.

2.1. Data model and types

Variables are used in Swift to name the local variables, arguments, and returns of a function. The outermost function in a Swift script (akin to "main" in C) is unique only in that the variables in its environment can be declared "global" to make them accessible to every other function in the script.

Each variable in a Swift script is declared to be of a specific type. The Swift type model is simple, with no concepts of inheritance or abstraction. There are three basic classes of data types: primitive, mapped, and collection.

The four primary *primitive types* are integer, float, string, and boolean values. Common operators are defined for primitive types, such as arithmetic, concatenation, and explicit conversion. (An additional primitive type, "external," is provided for manual synchronization; we do not discuss this feature here.)

Mapped types are used to declare data elements that refer (through a process called "mapping," described in Section 2.5) to files external to the Swift script. These files can then be read and written by application programs called by Swift. The mapping process can map single variables to single files, and structures and arrays to collections of files. The language has no built-in mapped types. Instead, users declare type names, with no other structure to denote any mapped type names desired (for example, `type file; type log;`).

A variable that is declared to be a mapped file is associated with a *mapper*, which defines (often through a dynamic lookup process) the file that is mapped to the variable.

Mapped type and collection type variable declarations can be annotated with a *mapping* descriptor that specifies the file(s) to be mapped to the Swift data element(s).

For example, the following lines declare `image` to be an *mapped file type* and a variable named `photo` of type `image`. Since `image` is a mapped file type, it additionally declares that the variable refers to a single file named `shane.jpeg`:

```
type image {};
image photo <"shane.jpeg">;
```

The notation `{}` indicates that the type represents a reference to a single *opaque* file, that is, a reference to an external object whose structure is opaque to the Swift script. For convenience such type declarations typically use the equivalent shorthand `type image;` (This compact notation can be confusing at first but has become an accepted Swift idiom.)

The two *collection types* are structures and arrays. A *structure type* lists the set of elements contained in the structure, as for example in the following definition of the structure type `snapshot`:

```
type image;
type metadata;
type snapshot {
   metadata m;
   image i;
}
```

Members of a structure can be accessed by using the `.` operator:

```
snapshot sn;
image im;
im = sn.i;
```

Structure fields can be of any type, whereas arrays contain values of only a single type. Both structures and arrays can contain members of primitive, mapped, or collection types. In particular, arrays can be nested to provide multidimensional indexing.

The size of a Swift array is not declared in the program but is determined at run time, as items are added to the array. This feature proves useful for expressing some common classes of parallel computations. For example, we may create an array containing just those experimental configurations that satisfy a certain criterion. An array is considered "closed" when no

further statements that set an element of the array can be executed. This state is recognized at run time by information obtained from compile-time analysis of the script's call graph. The set of elements that is thus defined need not be contiguous; in the words, the index set may be sparse. As we will demonstrate below, the `foreach` statement makes it easy to access all elements of an array.

2.2. Built-in, application interface, and compound functions

Swift's *built-in functions* are implemented by the Swift runtime system and perform various utility functions (numeric conversion, string manipulation, etc.). Built-in operators (+, *, etc.) behave similarly.

An *application interface function* (declared by using the `app` keyword) specifies both the interface (input files and parameters; output files) of an application program and the command-line syntax used to invoke the program. It thus provides the information that the Swift runtime system requires to invoke that program in a location-independent manner.

For example, the following application interface defines a Swift function `rotate` that uses the common image processing utility `convert` [5] to rotate an image by a specified angle. The `convert` executable will be located at run time in a catalog of applications or through the `PATH` environment variable.)

```
app (image output) rotate(image input, int angle) {
   convert "-rotate" angle @input @output;
}
```

Having defined this function, we can now build a complete Swift script that rotates a file `puppy.jpeg` by 180 degrees to generate the file `rotated.jpeg`:

```
type image;
image photo <"puppy.jpeg">;
image rotated <"rotated.jpeg">;

app (image output) rotate(image input, int angle) {
   convert "-rotate" angle @input @output;
}

rotated = rotate(photo, 180);
```

The last line in this script looks like an ordinary function invocation. However, thanks to the application interface function declaration and the semantics of Swift, its execution in fact invokes the `convert` program, with variables on the left of the assignment bound to the output parameters and variables to the right of the function invocation passed as inputs.

This script can be invoked from the command line, as in the following example, in which Swift executes a single `convert` command, while automatically performing for the user features such as remote multisite execution and fault tolerance, as discussed later.

```
$ ls *.jpeg
shane.jpeg
$ swift example.swift
...
$ ls *.jpeg
shane.jpeg rotated.jpeg
```

A third class of Swift functions, the *compound function*, invokes other functions. For example, the following script defines a compound function `process` that invokes functions `first` and `second`. (A temporary file, `intermediate`, is used to connect the two functions. Since no mapping is specified, Swift generates a unique file name.)

```
(file output) process (file input) {
   file intermediate;
   intermediate = first(input);
   output = second(intermediate);
 }
```

This function is used in the following script to process a file `x.txt`, with output stored in file `y.txt`.

```
file x <"x.txt">;
file y <"y.txt">;
y = process(x);
```

Compound functions can also contain flow-of-control statements (described below), while application interface functions cannot (since the latter serve to specify the functional interface for a single application invocation).

2.3. Arrays and parallel execution

Arrays are declared by using the `[]` suffix. For example, we declare here an array containing three strings and then use the `foreach` construct to apply a function `analyze` to each element of that array. (The arguments `fruit` and `index` resolve to the value of an array element and that element's index, respectively.)

```
string fruits[] = {"apple", "pear", "orange"};
file tastiness[];
foreach fruit, index in fruits {
  tastiness[index] = analyze(fruit);
}
```

The `foreach` construct is a principal means of expressing concurrency in Swift. The body of the `foreach` is executed in parallel for every element of the array specified by the `in` clause. In this example, all three invocations of the `analyze` function are invoked concurrently.

2.4. Execution model: Implicit parallelism

We have now described almost all the Swift language. While Swift also provides conditional execution through the `if` and `switch` statements and explicit sequential iteration through the `iterate` statement, we don't elaborate on these, as they are less relevant to our focus here on Swift's parallel aspects.

The Swift execution model is based on a simple, uniform model. Every data object in Swift is built up from atomic data elements that contain three fields: a value, a state, and a queue of function invocations that are waiting for the value to be set.

Swift data elements (atomic variables and array elements) are *single-assignment*. They can be assigned at most one value during execution, and they behave as futures. This semantic provides the basis for Swift's model of parallel function evaluation and dependency chaining. While Swift collection types (arrays and structures) are not single-assignment, each of their elements is single-assignment.

Through the use of futures, functions become executable when their input parameters have all been set, either from existing data or from prior function executions. Function calls may be chained by passing an output variable of one function as the input variable to a second function. This dataflow-driven

model means that Swift functions are not necessarily executed in source-code order but rather when their input data become available.

Since all variables and collection elements are single-assignment, a function or expression can be executed when all of its input parameters have been assigned values. As a result of such execution, more variables may become assigned, possibly allowing further parts of the script to execute. In this way, scripts are implicitly concurrent.

For example, in the following script fragment, execution of functions `p` and `q` can occur in parallel:

```
y=p(x);
z=q(x);
```

while in the next fragment, execution is serialized by the variable `y`, with function `p` executing before `q`:

```
y=p(x);
z=q(y);
```

Note that reversing the order of these two statements in a script will not affect the order in which they are executed.

Statements that deal with the array as a whole will wait for the array to be closed before executing. An example of such an action is the expansion of the array values into an `app` function command line. Thus, the closing of an array is the equivalent to setting a future variable, with respect to any statement that was waiting for the array itself to be assigned a value. However, a `foreach` statement will apply its body of statements to elements of an array in a fully concurrent, pipelined manner, as they are set to a value. It will not wait until the array is closed. In practice this type of "pipelining" gives Swift scripts a high degree of parallelism at run time.

The simplicity and regularity of the Swift data model make it easy to achieve a high degree of implicit parallelism. For example, a `foreach()` statement that processes an array returned by a function may begin processing members of the returned array that have been already set, even before the entire function completes and returns. The result is often programs that are heavily pipelined with significant overlapping parallel activities.

Consider the script below:

```
file a[];
file b[];
```

```
foreach v,i in a {
   b[i] = p(v);
}
a[0] = r();
a[1] = s();
```

Initially, the `foreach` statement will block, with nothing to execute, since the array `a` has not been assigned any values. At some point, in parallel, the functions `r` and `s` will execute. As soon as either of them is finished, the corresponding invocation of function `p` will occur. After both `r` and `s` have completed, the array `a` will be regarded as closed, since no other statements in the script make an assignment to `a`.

2.5. Swift mappers

Swift provides an extensible set of built-in mapping primitives ("mappers") that make a given variable name refer to a filename. A mapper associated with a structured Swift variable can represent a large, structured data set. A representative set of built-in mappers is listed in Table 1. Collections of files can be mapped to complex types (arrays and structures) by using a variety of built-in mappers. For example, the following declaration

```
file frames[] <filesys_mapper; pattern="*.jpeg">;
foreach f,ix in frames {
  output[ix] = rotate(f, 180);
}
```

uses the built-in `filesys_mapper` to map all files matching the name pattern `*.jpeg` to an array–and then applies a function to each element of that array.

Swift mappers can operate on files stored on the local machine in the directory where the `swift` command is executing, or they can map any files accessible to the local machine, using absolute pathnames. Custom mappers (and some of the built-in mappers) can also map variables to files specified by URIs for access from remote servers via protocols such as GridFTP or HTTP, as described in Section 3. Mappers can interact with structure fields and array elements in a simple and useful manner.

New mappers can be added to Swift either as Java classes or as simple, external executable scripts or programs coded in any language. Mappers can operate both as input mappers (which map files to be processed as application inputs) and as output mappers (which specify the names of files to

Table 1: Example of selected built-in mappers showing their syntax and semantics

Mapper Name	Description	Example

`single_file_mapper`	maps single named file	`file f <"data.txt">;`
		—
		`f` → `data.txt`

`filesys_mapper`	maps directory contents into an array	`file f[] <filesys_mapper;`
		`prefix="data",`
		`suffix=".txt">;`
		—
		`f[0]` → `data2.txt`

`simple_mapper`	maps components of the variable name	`file f <simple_mapper;`
		`prefix="data.",`
		`suffix=".txt">;`
		—
		`f.red` → `data.red.txt`

be produced by applications). It is important to understand that mapping a variable is a different operation from setting the value of a variable. Variables of mapped-file type are mapped (conceptually) when the variable becomes "in scope," but they are set when a statement assigns them a value. Mapper invocations (and invocations of external mapper executables) are completely synchronized with the Swift parallel execution model.

This ability to abstract the processing of files by programs as if they were in-memory objects and to process them with an implicitly parallel programming model is Swift's most valuable and noteworthy contribution.

2.6. Swift application execution environment

A Swift `app` declaration describes how an application program is invoked. In order to provide a consistent execution environment that works for virtually all application programs, the environment in which programs are executed needs to be constrained with a set of conventions. The Swift execution model is based on the following assumptions: a program is invoked in its own working directory; in that working directory or one of its subdirectories, the program can expect to find all files passed as inputs to the application block; and on exit, it should leave all files named by that application block in the same working directory.

Applications should not assume that they will be executed on a particular host (to facilitate site portability), that they will run in any particular order with respect to other application invocations in a script (except those implied by data dependency), or that their working directories will or will not be cleaned up after execution. In addition, applications should strive to avoid side effects that could limit both their location independence and the determinism (either actual or de facto) of the overall results of Swift scripts that call them.

Consider the following `app` declaration for the `rotate` function:

```
app (file output) rotate(file input, int angle)
```

The function signature declares the inputs and outputs for this function. As in many other programming languages, this declaration defines the type signatures and names of parameters; this also defines which files will be placed in the application working directory before execution and which files will be expected there after execution. For the above declaration, the file mapped to the `input` parameter will be placed in the working directory beforehand, and the file mapped to `output` will be expected there after execution; since the input parameter `angle` is of primitive type, no files are staged in for this parameter.

The body of the `app` block defines the command line that will be executed when the function is invoked:

```
convert "-rotate" angle @input @output;
```

The first token (in this case `convert`) defines a *application name* that is used to locate the executable program. Subsequent expressions define the command-line arguments for that executable: "`-rotate`" is a string literal; `angle` specifies the value of the angle parameter; and the syntax `@variable` (shorthand for the built-in function `@filename()`) evaluates to the filename of the supplied variable. Thus `@input` and `@output` insert the filenames of the corresponding parameters into the command line. We note that the filename of `output` can be taken even though it is a return parameter; although the value of that variable has not yet been computed, the filename to be used for that value is already available from the mapper.

3. The Swift runtime environment

The Swift runtime environment comprises a set of services providing the parallel, distributed, and reliable execution that underlie the simple Swift

language model. A key contribution of Swift is the extent to which the language model has been kept simple by factoring the complexity of these issues out of the language and implementing them in the runtime environment. Notable features of this environment include the following:

- Location-transparent execution: automatic selection of a location for each program invocation and management of diverse execution environments. A Swift script can be tested on a single local workstation. The same script then can be executed on a cluster, one or more grids of clusters, or a large-scale parallel supercomputer such as the Sun Constellation [6] or the IBM Blue Gene/P [7].

- Automatic parallelization of application program invocations that have no data dependencies. The pervasive implicit parallelism inherent in a Swift script is made practical through various throttling and scheduling semantics of the runtime environment.

- Automatic balancing of work over available resources, based on adaptive algorithms that account for both resource performance and reliability and that throttle program invocations at a rate appropriate for each execution location and mechanism.

- Reliability, through automated replication of application invocations, automatic resubmission of failed invocations, and the ability to continue execution of interrupted scripts from the point of failure.

- Formalizing of the creation and management of data objects in the language and recording of the provenance of data objects produced by a Swift script.

Swift is implemented by generating and executing a Karajan program [8], which provides several benefits: a lightweight threading model, futures, remote job execution, and remote file transfer and data management. Both remote execution and data transfer and management functions are provided through abstract interfaces called *providers* [8]. *Data providers* enable data transfer and management to be performed through a wide variety of protocols including direct local copying, GridFTP, HTTP, WebDAV, SCP, and FTP. *Execution providers* enable job execution to take place by using direct POSIX process fork, Globus GRAM, Condor (and Condor-G), PBS,

SGE, and SSH services. The Swift execution model can be flexibly extended for novel and evolving computing environments by implementing new data providers and/or job execution providers.

3.1. Executing on a remote site

Given Swift's pragmatically constrained model of application invocation and file passing, execution of a program on a remote site is straightforward. The Swift runtime system must prepare a remote working directory for each job with appropriate input files staged in; next it must execute the program; and then it must stage the output files back to the submitting system. The execution site model used by Swift is shown in Figure 1.

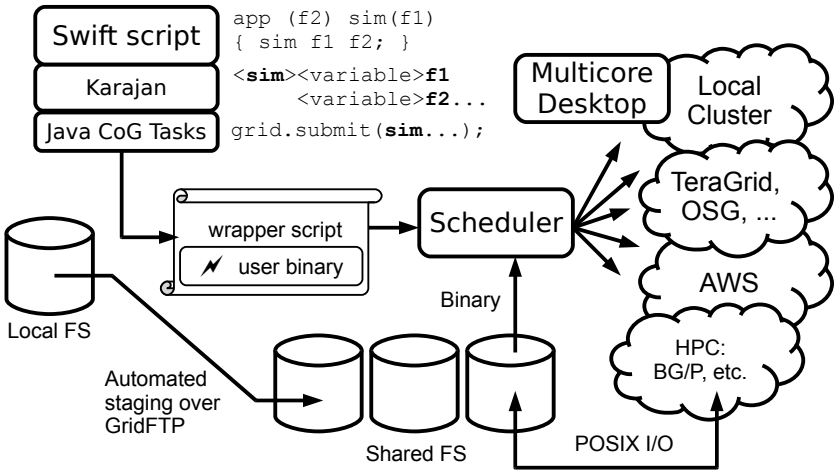


Figure 1: Swift site model (CoG = Commodity Grid [8], OSG = Open Science Grid, AWS = Amazon Web Services, HPC = high-performance computing system, BG/P = Blue Gene/P).

A site in Swift consists of one or more worker nodes, which will execute programs through some *execution provider*, and an *accessible file system*, which must be visible on the worker nodes as a POSIX-like file system and must be accessible to the Swift client command through some *file access provider*.

Two common implementations of this model are execution on the local system and execution on one or more remote clusters in a grid managed by Globus [9] software. In the former case, a local scratch file system (such as

`/var/tmp`) may be used as the accessible file system; execution of programs is achieved by direct POSIX fork, and access to the file system is provided by the POSIX filesystem API. (This approach enables Swift to make efficient use of increasingly powerful multicore computers.) In the case of a grid site, a shared file system is commonly provided by the site and is mounted on all its compute nodes; GRAM [10] and a local resource manager (LRM) provide an execution mechanism, and GridFTP [11] provides access from the submitting system to the remote file system.

Sites are defined and described in the *site catalog*:

```
<pool handle="tguc">
   <gridftp
      url="gsiftp://tg-gridftp.uc.teragrid.org" />
   <execution provider="gt2" jobmanager="PBS"
      url="tg-grid.uc.teragrid.org" />
   <workdirectory>
      /home/ben/swiftwork
   </workdirectory>
</pool>
```

This file may be constructed either by hand or mechanically from some pre-existing database (such as a grid's resource database interface). The site catalog is reusable and may be shared among multiple users of the same resources. This approach separates Swift application scripts from system configuration information and keeps the former location-independent.

The site catalog may contain definitions for multiple sites, in which case execution will be attempted on all sites. In the presence of multiple sites, it is necessary to choose between the available sites. To this end, the Swift *site selector* maintains a score for each site that determines the load that Swift will place on that site. As a site succeeds in executing jobs, this score is increased; as job executions at a site fail, this score is decreased. In addition to selecting between sites, this mechanism provides a measure of dynamic rate limiting if sites fail because of overload [12].

This dynamically fluctuating score provides an empirically measured estimate of a site's ability to bear load, distinct from and more relevant to scheduling decisions than is static configuration information. For example, site policies restricting job counts are often not available or accurate. In addition, a site's capacity or resource availability may not be properly quan-

tified by published information, for example, because of load caused by other users.

3.2. Reliable execution

The functional nature of Swift provides a clearly defined interface to imperative components, which, in addition to allowing Swift great flexibility in where and when it runs application programs, allows those imperative components to be treated as atomic components that can be executed multiple times for any given Swift function invocation. This facilitates three different reliability mechanisms that are implemented by the runtime system and that need not be exposed at the language level: *retries*, *restarts*, and *replication*.

In the simplest form of error handling in Swift, if an application program fails, Swift will attempt to rerun the program. In contrast to many other systems, retry here is at the level of the Swift function invocation and includes completely reattempting site selection, stage-in, execution, and stage-out. This provides a natural way to deal both with many transient errors, such as temporary network loss, and with many changes in site state.

Some errors are more permanent; for example, an application program may have a bug that causes it to always fail when given a particular set of inputs. In this case, Swift's retry mechanism will not help; each job will be tried a number of times, and each will fail, ultimately resulting in the entire script failing.

In such a case, Swift provides a *restart log* that encapsulates which function invocations have been successfully completed. A subsequent Swift run may be started with this restart log; this will avoid re-execution of already executed invocations.

A different class of failure occurs when jobs are submitted to a site and then remain enqueued for an extended time on that site. This is a "failure" in site selection, rather than in execution. It can be either a soft failure, where the job will eventually run on the chosen site (the site selector has improperly chosen a heavily loaded site), or a hard failure, where the job will never run because a site has ceased to process jobs of some class (or has halted all processing).

To address this situation, Swift provides for *job replication*. After a job has been queued on a site for too long (based on a configurable threshold), a replica of the job will be submitted (again undergoing independent site selection, staging, and execution); this will continue up to a defined limit. When any one of those jobs begins executing, all other replicas of the job

will be canceled. This replication algorithm nicely handles the "long tail" of "straggler jobs" [13, 14] that often delays completion of a parallel workload.

3.3. Avoiding job submission penalties

In many applications, the overhead of job submission through commonly available mechanisms, such as through GRAM into an LRM, can dominate the execution time. The reason is that the overhead of remote job submission may be long relative to the job length or that the job may wait in a congested queue, or both. In these situations, it is helpful to combine a number of Swift-level application program executions into a single GRAM/LRM submission.

Swift offers two mechanisms to address this problem: *clustering* and *coasters*. Clustering aggregates multiple program executions into a single job, thereby reducing the total number of jobs to be submitted. Coasters [15] is a form of multilevel scheduling similar to pilot jobs [16]. It submits generic coaster jobs to a site and binds component program executions to the coaster jobs (and thus to worker nodes) as these coaster jobs begin remote execution.

Clustering requires little additional support on the remote site, while the coasters framework requires an active component on the head node (in Java) and on the worker nodes (in Perl) as well as additional network connectivity within a site. Occasionally, the automatic deployment and execution of the coaster components can be problematic or even impractical on a site and may require alternative manual configuration.

However, clustering can be less efficient than using coasters. Coasters can react much more dynamically to changing numbers of available worker nodes. Clustering requires an estimate of available remote node count and job duration to decide on a sensible cluster size. Incorrectly estimating this can, in one direction, result in an insufficient number of worker nodes, with excessive serialization, or, in the other direction, result in an excessive number of job submissions. Coaster workers can be queued and executed before all of the work that they will eventually execute is known; hence, the Swift scheduler can perform more application invocations per coaster worker job and thus achieve faster overall execution of the entire application.

With coasters, the status for an application job is reported when the job actually starts and ends; with clustering, a job's completion status is known only when the entire cluster of jobs completes. This means that subsequent activity (stage-outs and, more important, starting dependent jobs) is delayed; in the worst case, an activity dependent on the first job in a cluster must wait for all of the jobs to run.

3.4. Features to support use of dynamic resources

Using Swift to submit to a large number of sites poses a number of practical challenges that are not encountered when running on a small number of sites. These challenges are seen when comparing execution on the relatively static TeraGrid [17] with execution on the more dynamic Open Science Grid (OSG) [18], where the set of sites that may be used is large and changing. It is impractical to maintain a site catalog by hand in this situation. In collaboration with the OSG Engagement group, Swift has been interfaced to ReSS [19], an OSG resource selection service, so that the site catalog is generated from that information system. This provides a straightforward way to generate a large catalog of sites.

Having built a catalog, two significant problems remain: the quality of those sites may vary wildly, and user applications may not be installed on those sites. Individual OSG sites, for example, may exhibit extremely different behavior, both with respect to other sites at the same time and with respect to themselves at other times. The load that a particular site will bear varies over time, and sites can fail in unusual ways. Swift's site scoring mechanism deals well with this situation in the majority of cases. However, discoveries of new and unusual failure modes continue to drive the implementation of increasingly robust fault tolerance mechanisms.

When running jobs on dynamically discovered sites, it is likely that component programs are not installed on those sites. To deal with this situation, OSG Engagement has developed best practices, which are implemented straightforwardly in Swift. Applications may be compiled statically and deployed as a small number of self-contained files as part of the input for a component program execution; in this case, the application files are described as mapped input files in the same way as input data files and are passed as a parameter to the application executable. Swift's existing input file management then stages in the application files once per site per run.

4. Applications

By providing a minimal language that allows the rapid composition of existing executable programs and scripts into a logical unit, Swift has become a beneficial resource for small to moderate-sized scientific projects.

Swift has been used to perform computational biochemical investigations, such as protein structure prediction [20, 21, 22], molecular dynamics simulations of protein-ligand docking [23] and protein-RNA docking, and searching

mass-spectrometry data for posttranslational protein modifications [20, 24]; modeling of the interactions of climate, energy, and economics [20, 25]; post-processing and analysis of climate model results; explorations of the language functions of the human brain [26, 27, 28]; creation of general statistical frameworks for structural equation modeling [29]; and image processing for research in image-guided planning for neurosurgery [30].

This section describes in detail two representative Swift scripts from diverse disciplines. The first is a tutorial example (used in a class on data-intensive computing at the University of Chicago) that performs a simple analysis of satellite land-use imagery. The second script is taken (with minor formatting changes) directly from work done using Swift for an investigation into the molecular structure of glassy materials in the field of theoretical chemistry. In both examples, the intent is to show complete and realistic Swift scripts, annotated to make the nature of the Swift programming model clear and to provide a glimpse of real Swift usage.

4.1. Satellite image data processing.

The first example—Script 1 below—processes data from a large dataset of files that categorize the Earth's surface, derived from data from the MODIS sensor instruments that orbit the Earth on two NASA satellites of the Earth Observing System.

The dataset we use (for 2002, named `mcd12q1`) consists of 317 "tile" files that categorize every 250-meter square of non-ocean surface of the Earth into one of 17 "land cover" categories (for example, water, ice, forest, barren, urban). Each pixel of these data files has a value of 0 to 16, describing one square of the Earth's surface at a specific point in time. Each tile file has approximately 5 million 1-byte pixels (5.7 MB), covering 2400 x 2400 250-meter squares, based on a specific map projection.

The Swift script analyzes the dataset to select the top N files ranked by total area of specified sets of land-cover types. It then produces a new dataset with viewable color images of those selected data tiles. (A color-rendering step is required, since the input datasets are not viewable images; their pixel values are land-use codes.) A typical invocation of this script would be "*Find the top 12 urban tiles*" or "*Find the 16 tiles with the most forest and grassland.*" Since this script is used for tutorial purposes, the application programs it calls are simple shell scripts that use fast, generic image-processing applications to process the MODIS data. Thus, the example executes quickly while serving as a realistic tutorial script for much more

compute-intensive satellite data-processing applications.

The script is structured as follows. Lines 1–3 define three mapped file types; `MODISfile` for the input images, `landuse` for the output of the landuse histogram calculation, and `file` for any other generic file that we don't wish to assign a unique type to. Lines 7–32 define the Swift interface functions for the application programs `getLandUse`, `analyzeLandUse`, `colorMODIS`, `assemble`, and `markMap`.

Lines 36–41 use the built-in function `@arg()` to extract a set of science parameters from the `swift` command-line arguments with which the user invokes the script. (This is a keyword-based analog of C's `argv[]` convention.) These parameters indicate the number of files of the input set to select (to process the first M of N files), the set of land cover types to select, the number of "top" tiles to select, and the input and output directories.

Lines 47–48 invoke an "external" mapper script `modis.mapper` to map the first `nFiles` MODIS data files in the directory contained in the script argument `MODISdir` to the array `geos`. An external mapper script is written by the Swift programmer (in any language desired, but often mappers are simple shell scripts). External mappers are usually colocated with the Swift script and are invoked when Swift instantiates the associated variable. They return a two-field list of the the form *SwiftExpression, filename*, where *SwiftExpression* is relative to the variable name being mapped. For example, if this mapper invocation were called from the Swift script at lines 47–48:

```
$ ./modis.mapper -location /home/wilde/modis/2002/ -suffix .tif -n 5
[0] /home/wilde/modis/2002/h00v08.tif
[1] /home/wilde/modis/2002/h00v09.tif
[2] /home/wilde/modis/2002/h00v10.tif
[3] /home/wilde/modis/2002/h01v07.tif
[4] /home/wilde/modis/2002/h01v08.tif
```

it would cause the first five elements of the array `geos` to be mapped to the first five files of the modis dataset in the specified directory.

At lines 52–53, the script declares the array `land`, which will contain the output of the `getlanduse` application. This declaration uses the built-in "structured regular expression mapper," which will determine the names of the *output* files that the array will refer to once they are computed. Swift knows from context that this is an output mapping. The mapper will use regular expressions to base the names of the output files on the filenames of the corresponding elements of the input array `geos` given by the `source=` argument to the mapper. The declaration for `land[]` maps, for example,

a file `h07v08.landuse.byfreq` to an element of the `land[]` array for a file `h07v08.tif` in the `geos[]` array.

At lines 55–57 the script performs its first computation using a `foreach` loop to invoke `getLandUse` in parallel on each file mapped to the elements of `geos[]`. Since 317 files were mapped (in lines 47–48), the loop will submit 317 instances of the application in parallel to the execution provider. These will execute with a degree of parallelism subject to available resources. At lines 52–53 the result of each computation is placed in a file mapped to the array `land` and named by the regular expression translation based on the file names mapped to `geos[]`. Thus the landuse histogram for file `h00v08.tif` would be written into file `h00v08.landuse.freq` and would be considered by Swift to be of type `landuse`.

Once all the land usage histograms have been computed, the script executes `analyzeLandUse` at line 63 to find the N tile files with the highest values of the requested land cover combination. Swift uses futures to ensure that this analysis function is not invoked until all of its input files have computed and transported to the computation site chosen to run the analysis program. All these steps take place automatically, using the relatively simple and location-independent Swift expressions shown. The output files to be used for the result are specified in the declarations at lines 61–62.

To visualize the results, the application function `markMap` invoked at line 68 will generate an image of a world map using the MODIS projection system and indicate the selected tiles matching the analysis criteria. Since this statement depends on the output of the analysis (`topSelected`), it will wait for the statement at line 63 to complete before commencing.

For additional visualization, the script assembles a full map of all the input tiles, placed in their proper grid location on the MODIS world map projection, and with the selected tiles marked. Since this operation needs true-color images of every input tile, these are computed—again in parallel—with 317 jobs generated by the foreach statement at lines 76–78. The power of Swift's implicit parallelization is shown vividly here: since the `colorMODIS` call at line 77 depends only on the input array `geos`, these 317 application invocations are submitted in parallel with the initial 317 parallel executions of the `getLandUse` application at line 56. The script concludes at line 83 by assembling a montage of all the colored tiles and writing this image file to a web-accessible directory for viewing.

Swift example 1: MODIS satellite image processing script

```
type file;
type MODIS; type image;
type landuse;

# Define application program interfaces

app (landuse output) getLandUse (imagefile input, int sortfield)
{
  getlanduse @input sortfield stdout=@output;
}

app (file output, file tilelist) analyzeLandUse
    (MODIS input[], string usetype, int maxnum)
{
  analyzelanduse @output @tilelist usetype maxnum @filenames(input);
}

app (image output) colorMODIS (MODIS input)
{
  colormodis @input @output;
}

app (image output) assemble
    (file selected, image img[], string webdir)
{
  assemble @output @selected @filename(img[0]) webdir;
}

app (image grid) markMap (file tilelist)
{
  markmap @tilelist @grid;
}

# Constants and command line arguments

int nFiles =      @toint(@arg("nfiles","1000"));
int nSelect =     @toint(@arg("nselect","12"));
string landType = @arg("landtype","urban");
string runID =    @arg("runid","modis-run");
string MODISdir=  @arg("modisdir","/home/wilde/bigdata/data/modis/2002");
string webDir =   @arg("webdir","/home/wilde/public_html/geo/");



# Input Dataset

image geos[] <ext; exec="modis.mapper",
  location=MODISdir, suffix=".tif", n=nFiles >;

# Compute the land use summary of each MODIS tile

landuse land[] <structured_regexp_mapper; source=geos, match="(h..v..)",
  transform=@strcat(runID,"/\\1.landuse.byfreq")>;
```

```
54
55  foreach g,i in geos {
56      land[i] = getLandUse(g,1);
57  }
58
59  # Find the top N tiles (by total area of selected landuse types)
60
61  file topSelected <"topselected.txt">;
62  file selectedTiles <"selectedtiles.txt">;
63  (topSelected, selectedTiles) = analyzeLandUse(land, landType, nSelect);
64
65  # Mark the top N tiles on a sinusoidal gridded map
66
67  image gridMap <"markedGrid.gif">;
68  gridMap = markMap(topSelected);
69
70  # Create multi-color images for all tiles
71
72  image colorImage[] <structured_regexp_mapper;
73            source=geos, match="(h..v..)",
74            transform="landuse/\\1.color.png">;
75
76  foreach g, i in geos {
77    colorImage[i] = colorMODIS(g);
78  }
79
80  # Assemble a montage of the top selected areas
81
82  image montage <single_file_mapper; file=@strcat(runID,"/","map.png") >; # @arg
83  montage = assemble(selectedTiles,colorImage,webDir);
```

4.2. Simulation of glass cavity dynamics and thermodynamics

Many recent theoretical chemistry studies of the glass transition in model systems have focused on calculating from theory or simulation what is known as the mosaic length. Glen Hocky of the Reichman Group at Columbia is evaluating a new cavity method [31] for measuring this length scale, where particles are simulated by molecular dynamics or Monte Carlo methods within cavities having amorphous boundary conditions.

In this method, various correlation functions are calculated at the interior of cavities of varying sizes and averaged over many independent simulations to determine a thermodynamic length. Hocky is using simulations of this method to investigate the differences between three glass systems that all have the same structure but differ in subtle ways; the aim is to determine whether this thermodynamic length causes the variations among the three systems.

The glass cavity simulation code performs 100,000 Monte Carlo steps in 1–2 hours. Jobs of this length are run in succession and strung together to

make longer simulations tractable across a wide variety of parallel computing systems. The input data to each simulation is a file of about 150 KB representing initial glass structures and a 4 KB file describing which particles are in the cavity. Each simulation returns three new structures of 150 KB each, a 50 KB log file, and the same 4 KB file describing which particles are in the cavity.

Each script run covers a simulation space of 7 radii by 27 centers by 10 models, requiring 1,890 jobs per run. Three model systems are investigated for a total of 90 runs. Swift mappers enable metadata describing these aspects to be encoded in the data files of the simulation campaigns to assist in managing the large volume of file data.

Hocky used four Swift scripts in his simulation campaign. The first, `glassCreate`, takes no input structure and generates an equilibrated configuration at some temperature; `glassAnneal` takes those structures and lowers the temperature to some specified temperature; `glassEquil` freezes particles outside a spherical cavity and runs short simulations for particles inside; and the script `glassRun`, described below, is the same but starts from equilibrated cavities.

Example 2 shows a slightly reformatted version of the glass simulation script that was in use in December 2010. Its key aspects are as follows. Lines 1–5 define the mapped file types; these files are used to compose input and output structures at lines 7–19. These structures reflect the fact that the simulation is restartable in one- to two-hour increments and that it works together with the Swift script to create a simple but powerful mechanism for managing checkpoint/restart across a long-running, large-scale simulation campaign.

The single application called by this script is the `glassRun` program wrapped in the app function at lines 21–29. Note that rather than defining main program logic in "open" (top-level) code, the script places all the program logic in the function `GlassRun`, invoked by the single statement at line 80. This approach enables the simulation script to be defined in a library that can be imported into other Swift scripts to perform entire campaigns or campaign subsets.

The `GlassRun` function starts by extracting a large set of science parameters from the Swift command line at lines 33–48 using the `@arg()` function. It uses the built-in function `readData` at lines 42–43 to read prepared lists of molecular radii and centroids from parameter files to define the primary physical dimensions of the simulation space. A selectable energy function to

be used by the simulation application is specified as a parameter at line 48.

At lines 57 and 61, the script leverages Swift flexible dynamic arrays to create a 3D array for input and a 4D array of structures for outputs. These data structures, whose leaf elements consist entirely of mapped files, are set by using the external mappers specified for the input array at lines 57–59 and for the output array of structures at lines 61–63. Note that many of the science parameters are passed to the mappers, which in turn are used by the input mapper to locate files within the large, multilevel directory structure of the campaign and by the output mapper to create new directory and file naming conventions for the campaign outputs. The mappers apply the common, useful practice of using scientific metadata to determine directory and file names.

The entire body of the `GlassRun` is a four-level nesting of `foreach` statements at lines 65–77. These loops perform a parallel parameter sweep over all combinations of radius, centroid, model, and job number within the simulation space. A single run of the script immediately expands to an independent parallel invocation of the simulation application for each point in the space: 1,890 jobs for the minimum case of a 7 x 27 x 10 x 1 space. Note that the `if` statement at line 69 causes the simulation execution to be skipped if it has already been performed, as determined by a "`NULL`" file name returned by the mapper for the output of a given job in the simulation space. In the current campaign the fourth dimension (`nsub`) of the simulation space is fixed at one. This value could be increased to define subconfigurations that would perform better Monte Carlo averaging, with a multiplicative increase in the number of jobs. This is currently set to one because there are ample starting configurations, but if this was not the case (as in earlier campaigns) the script could run repeated simulations with different random seeds.

The advantages of managing a simulation campaign in this manner are borne out well by Hocky's experience: the expression of the campaign is a well-structured, high-level script, devoid of details about file naming, synchronization of parallel tasks, location and state of remote computing resources, or explicit data transfer. Hocky was able to leverage local cluster resources on many occasions, but at any time he could count on his script's acquiring on the order of 1,000 compute cores from 6 to 18 sites of the Open Science Grid. When executing on the OSG, he leveraged Swift's capability to replicate jobs that were waiting in queues at more congested sites, and automatically sent them to sites where resources were available and jobs were being processed at better rates. All these actions would have represented a

huge distraction from his primary scientific simulation campaign if he had been required to use or to script lower-level abstractions where parallelism and remote distribution were the manual responsibility of the programmer.

Investigations of more advanced glass simulation techniques are under way, and the fact that the entire campaign can be driven by location-independent Swift scripts will enable Hocky to reliably re-execute the entire campaign with relative ease. He reports that Swift has made the project much easier to organize and execute. The project would be unwieldy without using Swift, and the distraction and scripting/programming effort level of leveraging multiple computing resources would be prohibitive.

Swift example 2: Monte-Carlo simulation of glass cavity dynamics

```
type Text;
type Arc;
type Restart;
type Log;
type Active;

type GlassIn{
    Restart startfile;
    Active activefile;
}

type GlassOut{
    Arc arcfile;
    Active activefile;
    Restart restartfile;
    Restart startfile;
    Restart final;
    Log logfile;
}

app (GlassOut o) glassCavityRun
    (GlassIn i, string rad, string temp, string steps, string volume, string fraca,
     string energyfunction, string centerstring, string arctimestring)
{   glassRun
      "-a" @filename(o.final) "--lf" @filename(i.startfile) stdout=@filename(o.logfile)
      "--temp" temp "--stepsperparticle" steps "--energy_function" energyfunction
      "--volume" volume "--fraca" fraca
      "--cradius" rad "--ccoord" centerstring arctimestring;
}

GlassRun()
{
  string temp=@arg("temp","2.0");
  string steps=@arg("steps","10");
  string esteps=@arg("esteps","100");
  string ceqsteps=@arg("ceqsteps","100");
  string natoms=@arg("natoms","200");
  string volume=@arg("volume","200");
  string rlist=@arg("rlist","rlist");
  string clist=@arg("clist","clist");
```

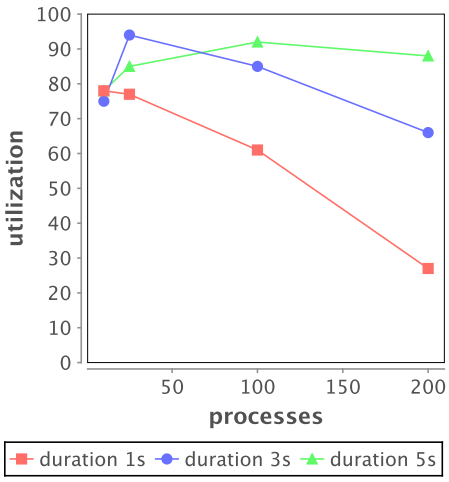
```
  string fraca=@arg("fraca","0.5");
  string radii[] = readData(rlist);
  string centers[] = readData(clist);
  int nmodels=@toint( @arg("n","1") );
  int nsub=@toint( @arg("nsub","1") );
  string savearc=@arg("savearc","FALSE");
  string arctimestring;
  string energyfunction=@arg("energyfunction","softsphereratiosmooth");

  if(savearc=="FALSE") {
    arctimestring="--arc_time=10000000";
  }
  else{
    arctimestring="";
  }

  GlassIn modelIn[][][] <ext; exec="GlassCavityOutArray.map",
    rlist=rlist, clist=clist, steps=ceqsteps, n=nmodels, esteps=esteps, temp=temp,
    volume=volume, e=energyfunction, natoms=natoms, i="true">;

  GlassOut modelOut[][][][] <ext; exec="GlassCavityContinueOutArray.map",
    n=nmodels, nsub=nsub, rlist=rlist, clist=clist, ceqsteps=ceqsteps, esteps=esteps,
    steps=steps, temp=temp, volume=volume, e=energyfunction, natoms=natoms>;

  foreach rad,rindex in radii {
    foreach centerstring,cindex in centers {
      foreach model in [0:nmodels-1] {
        foreach job in [0:nsub-1] {
          if( !(@filename(modelOut[rindex][cindex][model][job].final)=="NULL") ) {
            modelOut[rindex][cindex][model][job] = glassCavityRun(
              modelIn[rindex][cindex][model], rad, temp, steps, volume, fraca,
              energyfunction, centerstring, arctimestring);
          }
        }
      }
    }
  }
}

GlassRun();
```

5. Performance characteristics

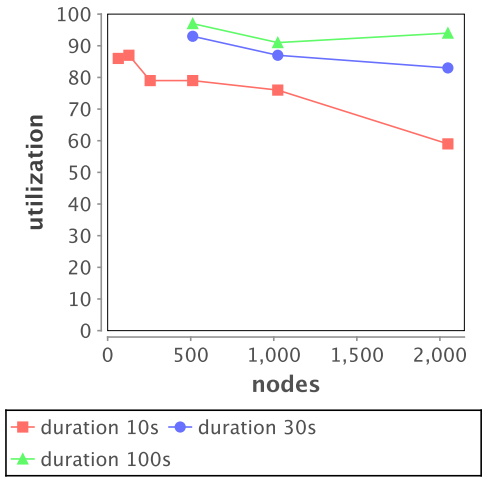
We present here a few additional measurements of Swift performance and highlight a few previously published results.

5.1. Synthetic benchmark results

First, we measured the ability of Swift to support many user tasks on a single local system. We used Swift to submit up to 2,000 tasks to a 16-core x86-based Linux compute server at Argonne National Laboratory. Each job in the batch was an identical, simple single-processor job that executed for the



Test A. Application CPU utilization for 3 task durations (in seconds) with up to 200 concurrent processes on an 16-core local host.



Test B. Application CPU utilization for 3 task durations (in seconds) at up to 2,048 nodes of the Blue Gene/P. at varying system size.

Figure 2: Swift performance figures

given duration and performed application input and output at 1 byte each. The total execution time was measured and compared with the total core time consumed; this utilization ratio is plotted in Figure 2, Test A. We observe that for tasks of only 5 seconds, Swift can sustain 100 concurrent application executions at a CPU utilization of 90%, and 200 concurrent executions at a utilization of 85%.

Second, we measured the ability of Swift to support many tasks on a large, distributed-memory system without considering the effect on the underlying file services. We used Swift coasters to submit up to 20,480 tasks to Intrepid, the 40,000-node IBM Blue Gene/P system at Argonne. Each job in the batch was an identical, simple single-processor job that executed for the given duration and performed no I/O. Each node was limited to one concurrent job. Thus, the user task had four cores at its disposal. The total execution time was measured and compared with the total node time consumed; the utilization ratio is plotted in Figure 2, Test B. We observe that for tasks of 100 seconds, Swift achieves a 95% CPU utilization of 2,048 compute nodes. Even for 30-second tasks, it can sustain an 80% utilization at this level of concurrency.

5.2. Application performance measurements

Previously published measurements of Swift performance on several scientific applications provide evidence that its parallel distributed programming model can be implemented with sufficient scalability and efficiency to make it a practical tool for large-scale parallel application scripting.

The performance of Swift submitting jobs over the wide-area network from the University of Chicago to the TeraGrid Ranger cluster at TACC is shown in Figure 3 (from [28]). The figure plots a structural equation modeling (SEM) workload of 131,072 jobs for four brain regions and two experimental conditions. This workflow completed in approximately 3 hours. The logs from the `swift_plot_log` utility show the high degree of concurrent overlap between job execution and input and output file staging to remote computing resources. The workflows were developed on and submitted (to Ranger) from a single-core Linux workstation at the University of Chicago running an Intel Xeon 3.20 GHz CPU. Data staging was performed by using the Globus GridFTP protocol, and job execution was performed over the Globus GRAM 2 protocol. During the third hour of the workflow, Swift achieved very high utilization of the 2,048 allocated processor cores and a steady rate of input and output transfers. The first two hours of the run were more bursty, because of fluctuating grid conditions and data server loads.

Prior work also attested to Swift's ability to achieve ample task rates for local and remote submission to high-performance clusters. These prior results are shown in Figure 4 (from [20]).

The top plot in Figure 4-A shows the PTMap application running the stage 1 processing of the *E. coli* K12 genome (4,127 sequences) on 2,048 Intrepid cores. The lower plot shows processor utilization as time progresses; overall, the average per task execution time was 64 seconds, with a standard deviation of 14 seconds. These 4,127 tasks consumed a total of 73 CPU-hours, in a span of 161 seconds on 2,048 processor cores, achieving 80% utilization.

The top plot in Figure 4-B shows the performance of Swift running a structural equation modeling problem at large scale, using the Ranger Constellation to model neural pathway connectivity from experimental fMRI data [28]. The lower plot shows the active jobs for a larger version of the problem type shown in Figure 3. This shows a Swift script executing 418,000 structural equation modeling jobs over a 40-hour period.

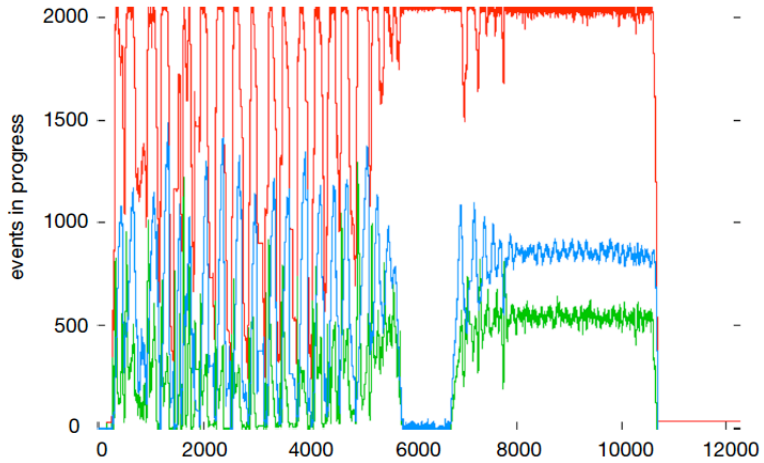
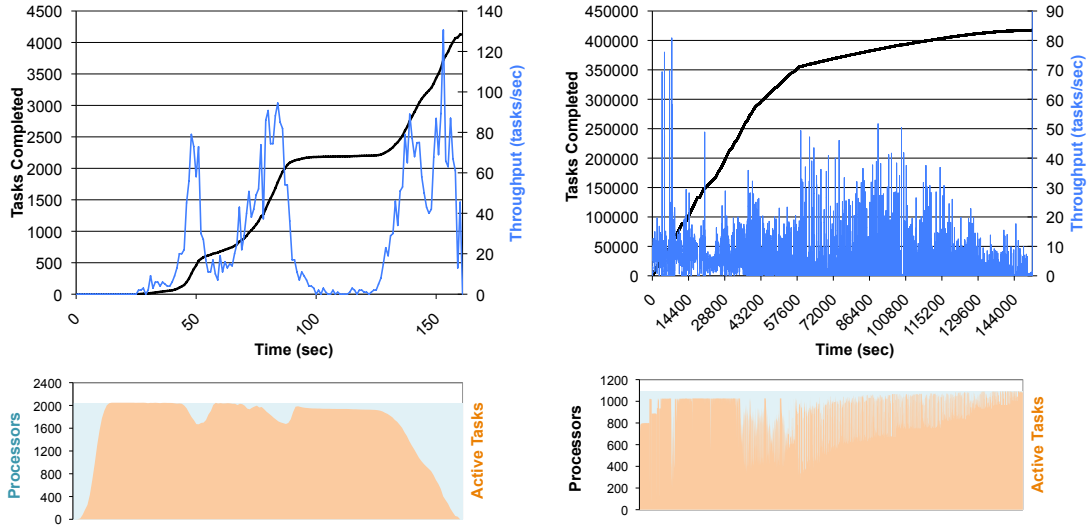


Figure 3: 128K-job SEM fMRI application execution on the Ranger Constellation (from [28]). Red=active compute jobs, blue=data stage in, green=stage out.



A. PTMap application on 2,048 nodes of the Blue Gene/P

B. SEM application on varying-size processing allocations on Ranger

Figure 4: Swift task rates for PTMap and SEM applications on the Blue Gene/P and Ranger (from [20])

6. Related Work

The rationale and motivation for scripting languages, the difference between programming and scripting, and the place of each in the scheme of applying computers to solving problems have previously been presented [32].

Coordination languages such as Linda [33], Strand [34], and PCN [35] support the composition of implicitly parallel functions programmed in specific languages and linked with the systems. In contrast, Swift coordinates the execution of distributed functions which are typically legacy applications which are coded in various programming languages, and can be executed on heterogeneous platforms. Linda defines primitives for concurrent manipulation of tuples ia a shared "tuple space". Strand and PCN, like Swift, use single-assignment variables as their coordination mechanism. Linda, Strand, PCN and Swift are all data-flow-driven in the sense that processes execute only when data are available.

MapReduce [13] also provides a programming model and a runtime system to support the processing of large-scale datasets. The two key functions *map* and *reduce* are borrowed from functional languages: a map function iterates over a set of items, performs a specific operation on each of them, and produces a new set of items; a reduce function performs aggregation on a set of items. The runtime system automatically partitions the input data and schedules the execution of programs in a large cluster of commodity machines. The system is made fault tolerant by checking worker nodes periodically and reassigning failed jobs to other worker nodes. Sawzall [36] is an interpreted language that builds on MapReduce and separates the filtering and aggregation phases for more concise program specification and better parallelization.

Swift and MapReduce/Sawzall share the same goals of providing a programming tool for the specification and execution of large parallel computations on large quantities of data and facilitating the utilization of large distributed resources. However, the two differ in many aspects. The MapReduce programming model supports key-value pairs as input or output datasets and two types of computation functions, map and reduce; Swift provides a type system and allows the definition of complex data structures and arbitrary computational procedures. In MapReduce, input and output data can be of several different formats, and new data sources can be defined; Swift provides a more flexible mapping mechanism to map between logical data structures and various physical representations. Swift does not automatically parti-

tion input datasets as MapReduce does; Swift datasets can be organized in structures, and individual items in a dataset can be transferred accordingly along with computations. MapReduce schedules computations within a cluster with a shared Google File System; Swift schedules across distributed grid sites that may span multiple administrative domains and deals with security and resource usage policy issues.

FlumeJava [37] is similar to Swift in concept, since it is intended to run data-processing pipelines over collections (of files). It is different in that it builds on top of MapReduce primitives, rather than more abstract graphs as in Swift.

BPEL [38] is a Web service-based standard that specifies how a set of Web services interact to form a larger, composite Web service. It has seen limited application in scientific contexts. While BPEL can transfer data as XML messages, for large datasets data exchange must be handled via separate mechanisms. The BPEL 1.0 specification provides no support for dataset iterations. An application with repetitive patterns on a collection of datasets could result in large, repetitive BPEL documents [39], and BPEL is cumbersome if not impossible for computational scientists to write. Although BPEL can use an XML Schema to describe data types, it does not provide support for mapping between a logical XML view and arbitrary physical representations.

DAGMan [40] provides a workflow engine that manages Condor jobs organized as directed acyclic graphs (DAGs) in which each edge corresponds to an explicit task precedence. It has no knowledge of data flow, and in a distributed environment it works best with a higher-level, data-cognizant layer. It is based on static workflow graphs and lacks dynamic features such as iteration or conditional execution, although these features are being researched.

Pegasus [41] is primarily a set of DAG transformers. Pegasus planners translate a workflow graph into a location-specific DAGMan input file, adding stages for data staging, intersite transfer, and data registration. They can prune tasks for existing files, select sites for jobs, and cluster jobs based on various criteria. Pegasus performs graph transformation with the knowledge of the whole workflow graph, while in Swift the structure of a workflow is constructed and expanded dynamically.

Dryad [42] is an infrastructure for running data-parallel programs on a parallel or distributed system. In addition to allowing files to be used for passing data between tasks (like Swift), it allows TCP pipes and shared-memory

FIFOs to be used. Dryad tasks are written in C++, whereas Swift tasks can be written in any language. Dryad graphs are explicitly developed by the programmer; Swift graphs are implicit, and the programmer doesn't have to worry about them. A scripting language called Nebula was originally developed above Dryad, but it doesn't seem to be in current use. Dryad appears to be used primarily for clusters and well-connected groups of clusters in single administrative domains and in Microsoft's cloud, whereas Swift supports a wider variety of platforms. Scripting-level use of Dryad is now supported primarily by DryadLINQ [43], which generates Dryad computations from the LINQ extensions to C#.

GEL [44] is somewhat similar to Swift. It defines programs to be run, then uses a script to express the order in which they should be run, handling the needed data movement and job execution for the user. The user must explicitly state what is parallel and what is not, whereas Swift determines this information based on data dependencies. GEL also lacks the runtime sophistication and platform support that has been developed for Swift.

Walker et al. [45] have recently developed extensions to BASH that allow a user to define a dataflow graph, including the concepts of fork, join, cycles, and key-value aggregation, but which execute on single parallel systems or clusters.

A few groups have been working on parallel and distributed versions of make [46, 47]. These tools use the concept of "virtual data," where the user defines the processing by which data is created and then calls for the final data product. The make-like tools determine what processing is needed to get from the existing files to the final product, which includes running processing tasks. If this is run on a distributed system, data movement also must be handled by the tools. In comparison, Swift is a language, which may be slightly less compact for describing applications that can be represented as static DAGs but allows easy programming of applications that have cycles and runtime decisions, such as in optimization problems. Moreover, Swift's functional syntax is a more natural companion for enabling the scientific user to specify the higher-level logic of large execution campaigns.

Swift integrates with the Karajan workflow engine [8]. Karajan provides the libraries and primitives for job scheduling, data transfer, and grid job submission. Swift adds to Karajan a higher-level abstract specification of large parallel computations and the typed data model abstractions of mapping disk-resident file structures to in-memory variables and data structures.

7. Future work

Swift is under active development. Current directions focus on improvements for short-running tasks, massively parallel resources, data access mechanisms, site management, and provenance.

7.1. Scripting on thousands to millions of cores

Systems such as the Sun Constellation [6] or IBM Blue Gene/P [7] have hundreds of thousands of cores, and systems with millions of cores are planned. Scheduling and managing tasks running at this scale are challenging problems and rely on the rapid submission of tasks. Swift applications currently run on these systems by scheduling Coasters workers using the standard job submission techniques and employing an internal IP network.

To achieve automatic parallelization in Swift, we ubiquitously use futures and lightweight threads, which result in eager and massive parallelism but which have a large cost in terms of space and internal object management. We are exploring several options to optimize this tradeoff and increase Swift scalability to ever larger task graphs. The solution space here includes "lazy futures" (whose computation is delayed until a value is first needed) and distributed task graphs with multiple, distributed evaluation engines running on separate compute nodes.

7.2. Filesystem access optimizations

Similarly, some applications deal with files that are uncomfortably small for GridFTP (on the order of tens of bytes). In this situation, a lightweight file access mechanism provided by Coasters can be substituted for GridFTP. When running on HPC resources, the thousands of small accesses to the filesystem may create a bottleneck for all system users. To mitigate this problem, we have investigated application needs and are developing a set of collective data management primitives [48].

7.3. Provenance

Swift produces log information regarding the provenance of its output files. In an existing development module, this information can be imported into relational and XML databases for later querying. Providing an efficient query mechanism for such provenance data is an area of ongoing research; while many queries can be easily and efficiently answered by a suitably indexed relational or XML database, the lack of support for efficient transitive

queries can make some common queries involving either transitivity over time (such as "Find all data derived from input file X") or over dataset containment (such as "Find all functions that took an input containing the file F") expensive to evaluate and awkward to express.

8. Conclusion

Our experience reinforces the belief that Swift plays an important role in the family of programming languages. Ordinary scripting languages provide the constructs for manipulating files and typically contain rich operators, primitives, and libraries for large classes of useful operations such as string, math, internet, and file operations. In contrast, Swift scripts typically contain little code that manipulates data directly. They contain instead the "data flow recipes" and input/output specifications of each program invocation such that the location and environment transparency goals can be implemented automatically by the Swift environment. This simple model has demonstrated many successes as a tool for scientific computing.

Swift is an open source project with documentation, source code, and downloads available at `http://www.ci.uchicago.edu/swift`.

Acknowledgments

This research was supported in part by NSF grants OCI-721939 and OCI-0944332 and by the U.S. Department of Energy under contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility, TeraGrid, the Open Science Grid, the UChicago/Argonne Computation Institute Petascale Active Data Store (PADS), and the Amazon Web Services Education allocation program.

The glass cavity simulation example in this article is the work of Glen Hocky of the Reichman Lab of the Columbia University Department of Chemistry. We thank Glen for his contributions to the text and code of Section 4 and valuable feedback to the Swift project. We gratefully acknowledge the contributions of current and former Swift team members, collaborators, and users: Sarah Kenny, Allan Espinosa, Zhao Zhang, Luiz Gadelha, David Kelly, Milena Nokolic, Jon Monette, Aashish Adhikari, Marc Parisien, Michael Andric, Steven Small, John Dennis, Mats Rynge, Michael Kubal, Tibi Stef-Praun, Xu Du, Zhengxiong Hou, and Xi Li. The initial implementation of

Swift was the work of Yong Zhao and Mihael Hategan; Karajan was designed and implemented by Hategan. We thank Tim Armstrong for helpful comments on the text.

References

[1] Haskell 98 Language and Libraries – The Revised Report, Internet document (2002).
URL http://haskell.org/onlinereport/haskell.html

[2] H. C. Baker, Jr., C. Hewitt, The incremental garbage collection of processes, in: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, ACM, New York, 1977, pp. 55–59. doi:http://doi.acm.org/10.1145/800228.806932.
URL http://doi.acm.org/10.1145/800228.806932

[3] A. D. Birrell, B. J. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems 2(1) (1984) 39–59.

[4] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, M. Wilde, Swift: Fast, Reliable, Loosely Coupled Parallel Computation, in: 2007 IEEE Congress on Services, 2007, pp. 199 –206. doi:10.1109/SERVICES.2007.63.

[5] ImageMagick project web site (2010).
URL http://www.imagemagick.org

[6] B.-D. Kim, J. E. Cazes, Performance and scalability study of Sun Constellation cluster 'Ranger' using application-based benchmarks, in: Proc. TeraGrid'2008, 2008.

[7] IBM Blue Gene team, Overview of the IBM Blue Gene/P project, IBM J. Res. Dev. 52 (2008) 199–220.
URL http://portal.acm.org/citation.cfm?id=1375990.1376008

[8] G. von Laszewski, M. Hategan, D. Kodeboyina, Java CoG kit workflow, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, 2007, Ch. 21, pp. 341–356.

[9] I. Foster, C. Kesselman, Globus: A metacomputing infrastructure toolkit, J. Supercomputer Applications 11 (1997) 115–128.

[10] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, A resource management architecture for metacomputing systems, in: D. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing, Vol. 1459 of Lecture Notes in Computer Science, Springer Berlin, 1998, pp. 62–82, 10.1007/BFb0053981.
URL http://dx.doi.org/10.1007/BFb0053981

[11] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster, The Globus striped GridFTP framework and server, in: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC '05, IEEE Computer Society, Washington, DC, 2005, pp. 54–. doi:10.1109/SC.2005.72.
URL http://dx.doi.org/10.1109/SC.2005.72

[12] D. Thain, M. Livny, The ethernet approach to grid computing, in: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, HPDC '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 138–.
URL http://portal.acm.org/citation.cfm?id=822087.823417

[13] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113. doi:10.1145/1327452.1327492.
URL http://doi.acm.org/10.1145/1327452.1327492

[14] T. Armstrong, M. Wilde, D. Katz, Z. Zhang, I. Foster, Scheduling many-task workloads on supercomputers: Dealing with trailing tasks, in: MTAGS 2010: 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers, 2010.

[15] M. Hategan, http://wiki.cogkit.org/wiki/Coasters.

[16] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, Condor-G: A computation management agent for multi-institutional grids, Cluster Computing 5 (2002) 237–246, 10.1023/A:1015617019423.
URL http://dx.doi.org/10.1023/A:1015617019423

[17] P. H. Beckman, Building the TeraGrid, Philosophical Transactions of the Royal Society A 363 (1833) (2005) 1715–1728. doi:10.1098/rsta.2005.1602.

[18] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, I. Foster, R. Gardner, M. Wilde, A. Blatecky, J. McGee, R. Quick, The Open Science Grid, Journal of Physics: Conference Series 78 (1) (2007) 012057.
URL http://stacks.iop.org/1742-6596/78/i=1/a=012057

[19] G. Garzoglio, T. Levshina, P. Mhashilkar, S. Timm, ReSS: A resource selection service for the Open Science Grid, in: S. C. Lin, E. Yen (Eds.), Grid Computing, Springer, N.Y., 2009, pp. 89–98, 10.1007/978-0-387-78417-5_8.
URL http://dx.doi.org/10.1007/978-0-387-78417-5_8

[20] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, I. Raicu, Parallel scripting for applications at the petascale and beyond, Computer 42 (11) (2009) 50–60. doi:10.1109/MC.2009.365.

[21] G. Hocky, M. Wilde, J. DeBartolo, M. Hategan, I. Foster, T. R. Sosnick, K. F. Freed, Towards petascale ab initio protein folding through parallel scripting, Tech. Rep. ANL/MCS-P1612-0409, Argonne National Laboratory (April 2009).

[22] J. DeBartolo, G. Hocky, M. Wilde, J. Xu, K. F. Freed, T. R. Sosnick, Protein structure prediction enhanced with evolutionary diversity: Speed, Protein Science 19 (3) (2010) 520–534.

[23] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford, Toward loosely coupled programming on petascale systems, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 22:1–22:12.
URL http://portal.acm.org/citation.cfm?id=1413370.1413393

[24] S. Lee, Y. Chen, H. Luo, A. A. Wu, M. Wilde, P. T. Schumacker, Y. Zhao, The first global screening of protein substrates bearing protein-bound 3,4-dihydroxyphenylalanine in Escherichia coli and human mitochondria., Journal of Proteome Research 9(11) (2010) 5705–5714.

[25] T. Stef-Praun, G. Madeira, I. Foster, R. Townsend, Accelerating solution of a moral hazard problem with Swift, in: e-Social Science 2007, Indianapolis, 2007.

[26] T. Stef-Praun, B. Clifford, I. Foster, U. Hasson, M. Hategan, S. L. Small, M. Wilde, Y. Zhao, Accelerating medical research using the Swift workflow system, Studies in Health Technology and Informatics 126 (2007) 207–216.

[27] U. Hasson, J. I. Skipper, M. J. Wilde, H. C. Nusbaum, S. L. Small, Improving the analysis, storage and sharing of neuroimaging data using relational databases and distributed computing, NeuroImage 39 (2) (2008) 693–706. doi:10.1016/j.neuroimage.2007.09.021.

[28] S. Kenny, M. Andric, S. B. M, M. Neale, M. Wilde, S. L. Small, Parallel workflows for data-driven structural equation modeling in functional neuroimaging, Frontiers in Neuroinformatics 3 (34). doi:10.3389/neuro.11/034.2009.

[29] S. Boker, M. Neale, H. Maes, M. Wilde, M. Spiegel, T. Brick, J. Spies, R. Estabrook, S. Kenny, T. Bates, P. Mehta, J. Fox, OpenMx: An open source extended structural equation modeling framework, Psychometrika In press.

[30] A. Fedorov, B. Clifford, S. K. Wareld, R. Kikinis, N. Chrisochoides, Non-rigid registration for image-guided neurosurgery on the TeraGrid: A case study, Tech. Rep. WM-CS-2009-05, College of William and Mary (2009).

[31] G. Biroli, J. P. Bouchaud, A. Cavagna, T. S. Grigera, P. Verrocchio, Thermodynamic signature of growing amorphous order in glass-forming liquids, Nature Physics 4 (2008) 771–775.

[32] J. Ousterhout, Scripting: Higher level programming for the 21st century, Computer 31 (3) (1998) 23–30. doi:10.1109/2.660187.

[33] S. Ahuja, N. Carriero, D. Gelernter, Linda and Friends, IEEE Computer 19(8) (1986) 26–34.

[34] I. Foster, S. Taylor, Strand: A practical parallel programming language, in: Proceedings of the North American Conference on Logic Programming, 1989, pp. 497–512.

[35] I. Foster, R. Olson, S. Tuecke, Productive parallel programming: The PCN approach, Sci. Program. 1 (1992) 51–66.
URL http://portal.acm.org/citation.cfm?id=1402583.1402587

[36] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, Interpreting the data: Parallel analysis with Sawzall, Scientific Programming 13 (4) (2005) 277–298.

[37] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, N. Weizenbaum, FlumeJava: Easy, efficient data-parallel pipelines, in: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 363–375. doi:10.1145/1806596.1806638.
URL http://doi.acm.org/10.1145/1806596.1806638

[38] M. B. Juric, Business Process Execution Language for Web Services, Packt Publishing, 2006.

[39] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, J. Patel, Sedna: A BPEL-based environment for visual scientific workflow modeling, in: I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, London, 2007, pp. 428–449, 10.1007/978-1-84628-757-2_26.
URL http://dx.doi.org/10.1007/978-1-84628-757-2_26

[40] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: The Condor experience, Concurrency and Computation: Practice and Experience 17 (2-4) (2005) 323–356. doi:10.1002/cpe.938.

[41] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gila, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, D. S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (2005) 219–237.

[42] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, in: Proceedings of European Conference on Computer Systems (EuroSys), 2007.

[43] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, J. Currey, DryadLINQ: A system for general-purpose distributed data-

parallel computing using a high-level language, in: Proceedings of Symposium on Operating System Design and Implementation (OSDI), 2008.

[44] C. Ching Lian, F. Tang, P. Issac, A. Krishnan, Gel: Grid execution language, J. Parallel Distrib. Comput. 65 (2005) 857–869. doi:10.1016/j.jpdc.2005.03.002.
URL http://dx.doi.org/10.1016/j.jpdc.2005.03.002

[45] E. Walker, W. Xu, V. Chandar, Composing and executing parallel dataflow graphs with shell pipes, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, WORKS '09, ACM, New York, 2009, pp. 11:1–11:10. doi:10.1145/1645164.1645175.
URL http://doi.acm.org/10.1145/1645164.1645175

[46] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, J. Tsujii, Design and implementation of GXP make – a workflow system based on make, in: Proceedings of IEEE International Conference on eScience, IEEE Computer Society, Los Alamitos, CA, 2010, pp. 214–221. doi:10.1109/eScience.2010.43.

[47] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, D. Thain, Harnessing parallelism in multicore clusters with the all-pairs, wavefront, and makeflow abstractions, Cluster Computing 13 (2010) 243–256, 10.1007/s10586-010-0134-7.
URL http://dx.doi.org/10.1007/s10586-010-0134-7

[48] J. M. Wozniak, M. Wilde, Case studies in storage access by loosely coupled petascale applications, in: Proceedings of the 4th Annual Workshop on Petascale Data Storage, PDSW '09, ACM, New York, 2009, pp. 16–20. doi:10.1145/1713072.1713078.
URL http://doi.acm.org/10.1145/1713072.1713078