Open access • Proceedings Article • DOI:10.1109/DFM.2014.17

Language Features for Scalable Distributed-Memory Dataflow Computing

Justin M. Wozniak, Michael Wilde, Ian Foster

Institutions: Argonne National Laboratory

Published on: 24 Aug 2014

Topics: Dataflow, Dataflow architecture, Signal programming, Specification language and Stream processing

Language Features for Scalable Distributed-Memory Dataflow Computing

Justin M. Wozniak,*† Michael Wilde,*† Ian T. Foster*†‡
*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
†Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA
‡Dept. of Computer Science, University of Chicago, Chicago, IL, USA

***Abstract*—Dataflow languages offer a natural means to express concurrency but are not a natural representation of the architectural features of high-performance, distributed-memory computers. When used as the outermost language in a hierarchical programming model, dataflow is very effective at expressing the overall flow of a computation. In this work, we present strategies and techniques used by the Swift dataflow language to obtain good performance on extremely large computing systems. We also present multiple unique language features that offer practical utility and performance enhancements.**

I. Introduction

Many applications are hierarchical– they consist of core performance-sensitive libraries, application-specific components, and high-level patterns such as statistics collection, parameter sweeps, or MapReduce-like structures. Many programmers attempt to solve problems at each different level with the same tool. In parallel and distributed computing, the outermost patterns are often expressed with a custom master-worker task distributor and blackboard system. The algorithms used may be elegantly expressed in dataflow format.

Swift/T [10] allows programmers to seamlessly and safely mix application logic with asynchronous task parallelism, using high-level data structures such as associative arrays and avoiding low-level concerns such as memory management. It offers familiar control flow statements, mathematical functions, and rich libraries for writing high-level "glue code" applications composing serial or parallel foreign functions written in languages such as C and Fortran, including calls to MPI libraries [11]. Swift applications executing tasks on CPUs, GPUs, or other devices can scale from multi-core workstations to high-performance computing (HPC) systems with hundreds of thousands of cores [5].

Even such seemingly trivial applications can require significant language expressiveness. A high-level language is perhaps the most intuitive and powerful way to express this kind of application logic. Ultimately, what many users want is a scripting language that lets them quickly develop scripts that compose high performance functions implemented in a native language. For sequential execution, dynamic languages such as shell scripts, Perl, or Python address this need. However, this paradigm breaks down when parallel computation is desired. With current sequential scripting languages, the logic must be rewritten and restructured to fit in a paradigm such as message passing, threading, or MapReduce. In contrast, Swift/T natively supports parallel and distributed execution while retaining the intuitive nature of sequential scripting, in which program logic is expressed directly with loops and conditionals.

Ahead-of-time compiler optimization and intelligent engineering of runtime systems are essential for this high-level programming model to be viable for applications that demand high performance. The goal of the Swift/T project is to develop an advanced compiler [2] and efficient runtime system [9] to implement the Swift language on extreme-scale, distributed-memory machines such as the Cray XE, IBM Blue Gene/Q, large clusters, and emerging systems in the exascale design space. In practice, this means translating the user-provided Swift program into an portable MPI program compatible with Linux-based and exotic HPC environments.

This work describes the Swift/T implementation and emphasizes its generalizable contributions to the dataflow computing community. In Section II, we provide background on the development of Swift. In Section III, we describe the Swift language as a dataflow language. In Section IV, we describe the MPI-based runtime libraries that implement Swift semantics. In Section V, we describe features unique to Swift/T of general interest to dataflow computing. In Section VI, we propose interesting dataflow programming features that may emerge in the Swift language. In Section VII, we summarize the paper and offer concluding remarks.

II. Background

The Swift language emerged from concerted efforts at the University of Chicago and elsewhere to produce a workflow language for grid computing, most recently resulting in an exascale-ready programming language. Early efforts (c. 2000) produced the Java Commodity Grid Kit (CoG) [8], which provided abstractions for remote execution. CoG included an optional Java-based directed-acyclic-graph (DAG) API. CoG then produced an XML-based programming language called GridAnt (c. 2002) [7], which had similarities to Apache Ant. A major revision of the GridAnt work resulted in Karajan (c. 2006) [1], which had both XML-based and functional syntaxes. Concurrently, the Virtual Data Language (VDL) projects (c. 2003) [4] produced pure "virtual data" tools for data-dependent processing on remote resources. The combined effort, VDL2 (c. 2006), produced a high-level dataflow language with C/Java syntax likenesses that translated into a

Karajan program. VDL2 was renamed to Swift (Scientific WorkFlow Tool) [12]. An exascale-funded project ExM (c. 2010) was launched to produce a version of Swift capable of running on exascale resources. Thus Swift was renamed to Swift/K (for Karajan), and the new system was named Swift/T (for the new Turbine runtime).

III. Overview of the Swift language

Superficially, Swift appears to be a simple, sequential language. It includes typical variable types, familiar control constructs, and typical arithmetic operators and builtin functions. It also includes *leaf functions*, which call into user code in the subordinate level of the programming hierarchy (commonly C or Fortran functions, or executables). As a dataflow language, however, all variables are *futures*; execution is based on data availability, not an instruction pointer.

```
1  int a, b, c, d;
2  a = A(input());
3  b = B(a);
4  c = C(a);
5  d = D(b,c);
6  output(d);
```

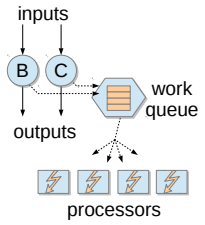

Fig. 1: Simple diamond dataflow pattern.

This is clearly a simple dataflow "diamond" pattern, in which `B()` and `C()` are eligible to run concurrently, as shown in Figure 1. If `B()` and `C()` are leaf functions, in the Swift model, each is submitted to a *work queue* that load balances and distributes the function as a task for remote execution.

Additional concurrency may be produced through the use of the `foreach` statement:

```
1  int S[] = [0:9];
2  foreach i in S {
3    int a, b, c, d;
4    a = A(input());
5    b = B(a);
6    c = C(a);
7    d = D(b,c);
8    output(d);
9  }
```

Example 1: Swift 'foreach' statement

In this fragment, the `[m:n]` syntax specifies a literal array of integers. For each entry `i`, a *block* is created, in which variables are dynamically created and dataflow begins.

Blocks themselves may be the subject of dataflow. In the `wait` statement, the block is specified as the object of dataflow evaluation, as shown in Figure 2.

```
1  int a, b, c, d;
2  a = A(input());
3  wait (a) {
4    b = B(F());
5  }
6  c = C(a);
7  d = D(b,c);
8  output(d);
```

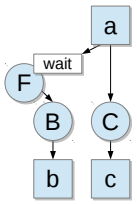

Fig. 2: Dataflow pattern with `wait` on externality.

We stress that `wait` is not required for dataflow computing-it is useful for certain "dirty hacks" that enable dependency on miscellaneous native computer features, such as the clock, changing external data sources, etc. It is also illustrative for more complex Swift constructs.

The `if` statement is essentially a `wait` plus a condition evaluation:

```
1  int a, b, c, d;
2  a = A(input());
3  boolean k = (a > 0);
4  if (k) { b = B(); }
5  else { b = 0; }
6  c = C(a);
7  d = D(b,c);
8  output(d);
```

Example 2: Swift 'if' statement

The two blocks that assign to `b` are equivalent to `wait(k)` blocks but are selected based on the value of the Boolean condition, `k`.

The combination of Swift conditional blocks and iteration enables the novel Swift/T `for` construct:

```
1  int d;
2  for (int i = 0; G(i,d); i = i+1) {
3    int a, b, c;
4    a = A(i);
5    b = B(a);
6    c = C(a);
7    d = D(b,c);
8    output(d);
9  }
```

Example 3: Swift 'for' statement

In this C-like construct, there is one key difference: the third statement in the `for` header (`i=i+1`) is *split* into left-hand and right-hand sides, which refer to variables (`i`) in different blocks (more than one such statement may be written, separated by commas). Thus, a data dependency from one iteration to the next is created. Across iterations, concurrent evaluation is possible depending on how dependencies are structured. The condition statement (`G(i,d)` is a conditional dependency for the next block. Thus, in the case shown, the next block depends on the value of `d` in the previous block, creating a strictly linear sequence of blocks, and limiting concurrency to within the block.

IV. Overview of the Swift runtime: Turbine

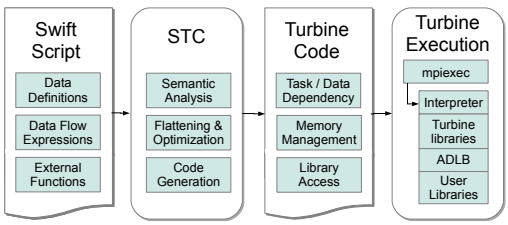

Fig. 3: Schematic of STC usage.

The Swift-Turbine Compiler (STC) is an optimizing compiler. The compiler interprets Swift syntax as an abstract syntax tree (AST). Based on user controls, it performs multiple optimization passes. It emits code in a representation compatible with our runtime, Turbine. This representation is a Tcl

script; thus, STC internally generates a Tcl AST and writes that to the generated file.

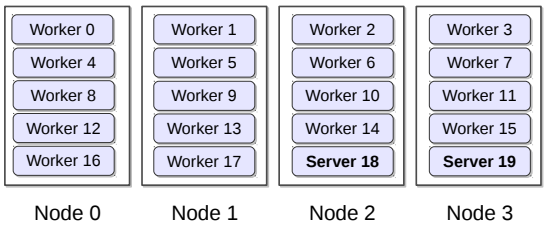

Fig. 4: Schematic of Turbine architecture.

The Turbine runtime is shown in Figure 4. Designed for massively multi-node systems, each node runs multiple Turbine processes that are launched using the system-specific MPI job scheduler (PBS, SLURM, etc.). Each process runs the STC-generated script. All communication is performed over the MPI-based Asynchronous Dynamic Load Balancer (ADLB) library [6]. Each process is dispatched into one of two execution modes based on rank number– worker or server.

A. Implementation

The original ADLB served as a scalable, lightweight library that offered two key RPCs- `Put()` and `Get()` on tasks. Tasks, each represented as a byte buffer (or message), are distributed for execution on workers and shared among servers based on memory limits. ADLB tasks have rich features, including priorities and location targeting. The ADLB library implements the work queue described previously.

For the Turbine effort, ADLB was significantly enhanced. Server work-sharing is now based on a Scioto-like [3] work stealing mechanism. Many additional RPCs were added to allow servers to manage data in addition to tasks, including data operations `Create()`, `Subscribe()`, `Store()`, and `Retrieve()`. Data creation allocates the memory location on a server in accordance with a simple hashing scheme. Data subscription results in a notification when data is stored, and data may be retrieved by any process. Thus, this creates a full-featured, write-once global data store with notifications.

B. Swift semantics

Our goal for Turbine was to create a convenient compiler target for STC. Thus, all ADLB operations (implemented in C) are exposed as Tcl statements, making up the (sequential) instruction set for *Turbine code*. The key statement for data-dependent, distributed execution is the Turbine *rule statement*:

`rule` $(v_0, v_1, ...)$ *continuation options...*

The rule establishes data dependencies on $v_0, v_1, ...$ and does not block. When these are resolved, the continuation is released to the work queue for processing. Options may be used to control details of how the continuation is executed.

A typical Swift expression is translated into Turbine code (shown as Tcl-like pseudocode) in Figure 5. As shown, the data-dependent block representing the implementation of `increment` is expressed as a Tcl `proc`. This implements the

```
1  int a;
2  a = 3;
3  int b;
4  b = increment(a);
5
6
7
8
9
10
```

Swift code

```
1  create integer a
2  store a 3
3  create integer b
4  rule (a) increment(b,a)
5
6  proc increment(b,a) {
7    retrieve a
8    a = a+1
9    store b a
10 }
```

Turbine code

Fig. 5: Elementary dataflow expression in Turbine

rule continuation on any worker process, by retrieving its state from the data store and storing its results, possibly releasing continuations elsewhere. Other blocks, such as those produced by `wait`, etc., may be translated in a similar manner, with appropriate data management.

C. A data-dependent master-worker system

At the lowest level (above MPI), the data-dependent task is presented to ADLB with the new `ADLB_Dput()` RPC, which allows a task submission to ADLB that is dependent on data availability. Thus, our system could support other frameworks that 1) use ADLB, 2) generate ADLB programs in the C language, or 3) translate to Swift. Our ADLB-level extensions offer a scalable, generic master-worker system with rich data features in addition to traditional task distribution.

V. Language features

In this section, we describe multiple novel features available in Swift/T for dataflow processing.

- **Prioritized evaluation ordering:** The priority annotation may be affixed to a Swift leaf function to modify the task priority in the ADLB work queue. Thus, the code:

```
1  int p = compute_priority(x);
2  output = @priority=p f(x);
```

Example 4: Task priorities

allows the user Swift function `compute_priority()` to set the priority of the task resulting from `f(x)`. This is an effective way for the user to control concurrency, by specifying high-priority tasks that satisfy many data dependencies early in execution. Another use is to run longer-running tasks as higher priority, allowing shorter tasks to fill in gaps at the end of a run, limiting the long-tail effect of an irregular run.

- **Task locations for data locality:** The location annotation may be affixed to a Swift leaf function to modify the task location setting in the ADLB. This targets a task to a certain rank. Swift/T provides builtin functions to translate hostnames to ranks, allowing the user to use data locations in an external storage system to drive execution. Thus the code:

```
1  foreach i in [0:9] {
2    string filename = "/fs/file-"+i;
3    string host = find_file(filename);
4    location L = host2rank(host);
5    @location=L compute(filename);
6  }
```

Example 5: Task locations

could be used to perform data location-aware scheduling, enabling data-intensive computing with Swift (provided a storage system that makes data locations available).

• **Updateable variables:** As described above, Swift execution occurs after the task has proceeded through the work queue. Thus, program state could have changed, invalidating work or changing requirements for work in the queue. Swift *updateables* allow Swift blocks to change the value of a variable. Thus, the receiving task must be capable of using the previous or updated value of the variable. In this example:

```
1 updateable int error = 0;
2 int c = f();
3 if (c < 0) { error := 1 };
4 output(@priority=LOW g(error));
```

Example 6: Updateable variables

task `g()` may or may not run concurrently with `f()`. If it does run later, it may have access to the latest value of `error`, allowing it to exit early or make other behavioral changes based on the latest value of `error`.

VI. Proposed language features

In this section, we propose new dataflow language features related to those described previously that may be implemented in Swift in the near future.

• **Compiler-managed data locality:** The Swift `@location` syntax provides low-level control to the programmer regarding task/data locality. We intend to partially automate this by annotating Swift data definitions with the `@heavy` annotation, enabling the compiler to generate appropriate `@location` directives. This would allow the user to influence the locality of data-intensive execution without the book-keeping required for manual data management.

• **Flexible task location targeting:** The ADLB API currently allows only two possibilities for task location: a specific MPI rank, or *anywhere*. This is not a good match for modern computer systems with complex data access costs. We propose to extend ADLB with soft targets, which will allow soft requirements for execution location, and expose these to the Swift programmer as we do with other annotations.

• **Optional data dependencies:** Updateables are a useful way to manage computation patterns that do not quite fit the dataflow model, yet push the limits of acceptable variations to dataflow processing. We propose a *soft dependency* feature that would allow work to be released to the work queue before soft dependencies are resolved. If the dependencies are resolved before task execution begins, the task would receive the user-stored value; otherwise a default value would be received:

```
1 int a = 10;
2 int d;
3 d = f(a);
4 output(@priority=LOW g(a, @soft d));
```

Example 7: Soft data dependencies

As shown in Example 7, the execution of `g()` may or may not receive the value of `d` set by `f()`, allowing maximal concurrency at the cost of determinism.

VII. Summary

In this work, we have described Swift from dataflow principles. We presented a brief history of the development of Swift from a convergence of distributed computing and dataflow computing concepts. We described how Swift uses dataflow to represent concurrent computation, including some of its syntactic structures for more complex dataflow and concurrency semantics. We also described how Swift may be translated into a representation compatible with our scalable dataflow-oriented runtime, Turbine. We then described some Swift-specific features to enhance dataflow computing in practice, including priority, locality, and non-deterministic extensions to dataflow computing. We intend that this will motivate future work in the dataflow community for solving extreme scale, high performance computing challenges with the elegance of scripted dataflow programs.

Acknowledgments

This research is supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357 and NSF award ACI 1148443. Computing resources were provided in part by NIH through the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory, under grant S10 RR029030-01, and by NSF award ACI 1238993 and the state of Illinois through the Blue Waters sustained-petascale computing project. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] Karajan manual. http://wiki.cogkit.org/index.php/Karajan.

[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] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Int'l Conf. on Parallel Processing*, 2008.

[4] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A new model and architecture for data-intensive collaboration. In *Proc. CIDR*, 2003.

[5] S. J. Krieder, J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. Design and evaluation of the GeMTC framework for GPU-enabled many task computing. In *Proc. HPDC*, 2014.

[6] 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, 2010.

[7] G. von Laszewski, B. Alunkal, K. Amin, S. Hampton, and S. Nijsure. GridAnt: Client-side workflow management with Ant. *Whitepaper*, 2002.

[8] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9), 2001.

[9] 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.

[10] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-scale application composition via distributed-memory data flow processing. In *Proc. CCGrid*, 2013.

[11] 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.

[12] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. Congress on Services*, 2007.