Swift/T: Dataflow Composition of Tcl Scripts for Petascale Computing

Justin M. Wozniak,[*†] Timothy G. Armstrong,[‡]
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

We present Swift/T, a novel programming language for extreme-scale supercomputers, that compiles into Tcl code for execution on a runtime based on Tcl and MPI. The Swift/T language enables users to rapidly compose Tcl fragments, including Tcl extension functions implemented in native code, into a massively scalable program capable of utilizing petaflop-scale computers. Swift/T has a growing user community in multiple scientific computing domains, including materials science, proteomics, and epidemiology. This paper summarizes the use of Swift/T for Tcl users, its performance achievements, and some early use cases.

I. Introduction

Modern supercomputers such as Cray XK7 Titan at Oak Ridge National Laboratory or IBM Blue Gene/Q Mira at Argonne National Laboratory demonstrate the massive concurrency emerging on extreme-scale supercomputers. Titan, totaling 20 petaflops, has 300 thousand CPU cores, plus 46 million CUDA cores; Mira has 768 thousand CPU cores. Managing this level of concurrency is extremely difficult with existing programming models such as the de facto standard toolkit: the C language with MPI messaging. To facilitate rapid prototyping and development, we propose the Swift/T language [1], which enables users to rapidly combine native code libraries and Tcl script fragments in a high-level scripting language. This eases the construction of computational experiments common in scientific computing.

Swift/T also runs on clusters and clouds and can be a powerful tool for achieving parallelism in smaller computing environments. The system uses Tcl in two key ways. First, Tcl is used as the compiler target for the Swift/T compiler. This provided a human-readable, high-level "ISA" for our system, allows us to easily call down to the elements of our runtime that are implemented in C, and allows us to send code fragments over the network. Second, Tcl can be accessed by the Swift/T application programmer to rapidly create scalable applications from Tcl packages that may link to Tcl extensions in native code. Swift/T thus creates a three-level application hierarchy [2] for massively parallel

scientific computing: Swift (for multi-node parallel dataflow) → Tcl (for package access and glue) → C/C++/Fortran (for performance-critical sequential computation).

This paper presents three key points. First, we describe the Swift/T language and explain how it achieves parallelism across nodes implicitly. Second, we describe the Tcl interfaces provided by Swift/T, and show how Swift/T can be used to write scalable Tcl programs. Finally, we describe use cases and performance results from previous work.

II. Parallelism in Swift/T

Swift/T is a functional dataflow language with a C-like syntax. Swift/T data types include integers, floats, strings, booleans, plus arrays, and structs of these types. All Swift/T data types are futures, write-once variables that enable data-dependent computation. For example, a line `y=f(x);` would block until `x` is written, then execute `f` and store `y` on completion, releasing any other statements blocked on `y`. At runtime, `f()` is treated as a task that is may be executed anywhere in the system; variable `x` is automatically transferred to `f()` and `y` is stored and made available for subsequent tasks.

Swift/T has no instruction pointer; like a declarative language all statements are issued to the runtime, then execute in dataflow order. Unlike other popular dataflow languages, Swift/T has conventional (C-like) programming constructs such as `if` conditions, `for` loops, and functions. However, these constructs are meant to operate at a high-level, and computationally intensive work is pushed into 'leaf' tasks that are opaque to Swift/T. Swift/T leaf functions are always Tcl fragments! These have been used to expose C/C++/Fortran libraries for Swift/T-level scripting (generally via SWIG [3]).

III. Swift/T interfaces for Tcl

Swift/T consists of a compiler (based on ANTLR) and a runtime called Turbine (based on ADLB [4] with Tcl bindings and supporting Tcl libraries). Tcl is the 'assembler language' of Swift/T. The Swift/T Compiler (STC) [5] translates a Swift/T script into Turbine Intermediate Code (TIC), which is a Tcl program. This program requires the Turbine package and runs in parallel over MPI. The TIC is platform-independent. It can make use of other user Tcl packages, which may use native code libraries via Tcl extensions.

As the 'assembler language', Tcl can be called from Swift/T (similarly to an `asm` block in C). Swift/T interfaces to Tcl contain a Swift/T type signature, Tcl package information (if a Tcl package must be loaded), and a Tcl 'leaf function' template. The template is filled in by STC with glue code to connect Swift/T input and output variables to Tcl. An example that computes $c = a + b$ is:

```
1 | (int c) add(int a, int b) "my_package" "0.0" [
2 |   "set <<c>> [ expr <<a>> + <<b>> ]"
3 | ];
```

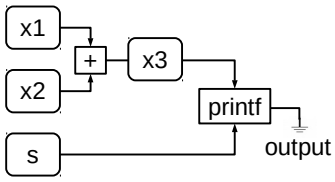
The C-like function signature indicates that all variables are integers (Swift/T is strictly typed). Swift/T variables are pasted into the Tcl template via the $<< \bullet >>$ syntax, and return values are copied out. Multiple lines of Tcl may be placed in the string (which can be formatted as a multi-line string), although it is preferable to call into a

Tcl library for more complex cases. Package `my_package` would be package-require'd once (regardless of how many functions use it). All Swift/T builtins are implemented using this technique!

IV. The Turbine runtime

The T in Swift/T stands for the Turbine runtime [6]. This is a parallel computing system built on Tcl, ADLB, and MPI. Consider this (runnable) example Swift code (`main.swift`) and its dataflow diagram:

```
1 import io;
2
3 x1 = 3;
4 s  = "value: ";
5 x2 = 2;
6 int x3;
7 printf("%s%i", s, x3);
8 x3 = x1+x2;
```



This code needs to register variables in distributed data store, and then perform operations on them in the future in data-dependent order. For example, the `printf()` statement (line 7) is issued to the runtime before `x3` is set (line 8). As shown, Swift/T data is typed, but STC can auto-detect types in many circumstances.

The (runnable) Turbine equivalent (`main.tic`) is shown below. Turbine allocates globally-accessible data with its `literal` and `allocate` statements. Then, it issues `rule` statements to implement concurrent, data-dependent execution. The first `rule` implements the `printf()`, the second implements the addition. (In practice, these are written using Turbine built-ins, avoiding the need to use the `\[\]` syntax to escape interpolation, but this example shows the explicit use of `rule`.) The type of Tcl variable `x1` is an integer, a Turbine Datum or TD for short. It is the address of the variable in the distributed data store. Data can be stored and retrieved by Tcl by referring to `$x1`.

```
1  package require turbine 0.0.1
2  namespace import turbine::*
3
4  proc main { } {
5      literal  x1 integer 3
6      literal  s  string  "value: "
7      literal  x2 integer 2
8      allocate x3 integer
9      rule [ list $x3 ] "puts \[retrieve $s\]\[retrieve $x3\]"
10     rule [ list $x1 $x2 ] \
11         "store_integer $x3 \[expr \[retrieve $x1\]+\[retrieve $x2\]\]"
12 }
13
14 turbine::defaults
15 turbine::init $servers "Swift"
16 turbine::start main
17 turbine::finalize
```

The rule statement arguments are the TD input list and an action string. The statement posts both arguments to an ADLB server that subscribes to each input TD. When all TDs are set, the action string is released to the work queue, and some worker obtains it via ADLB. It then simply performs an `eval` on that string. The action string can use the Turbine API to retrieve, allocate, or store data, perform any Tcl operations, and/or issue additional `rules`. Thus, a single worker can post a great deal of data-dependent

work to the system, and it will be executed as concurrently as possible preserving data dependencies.

This program can be run by setting `TCLLIBPATH` for the Turbine package and running

`mpiexec -n` *procs* `tclsh main.tic`

or using the provided Turbine wrapper program (no environment settings required)

`turbine -n` *procs* `main.tic`

or most simply

`swift-t -n` *procs* `main.swift`

which will invoke STC and then run the TIC program over *procs* MPI processes.

V. Swift/T use and performance

Swift/T is currently used for multiple scientific workflow applications, with varying performance requirements.

Managing ensembles of simulations is a core capability of Swift/T. Evolutionary algorithms can produce a large simulation workloads, and require a complex algorithm to manage the overall algorithm. In [7], we constructed an ensemble of epidemic simulations controlled by a third-party evolutionary algorithm (DEAP).

Workflows to analyze scientific data produced by large-scale materials science experiments, for example, can require very large computing resources operating on large data sets. In [8], we used Swift/T to load large data sets produced by an crystallographic X-ray detector onto a Blue Gene/Q, then execute existing C analysis code fragments wrapped in Tcl.

Swift/T can be used as a cluster data workflow system as well. In [9], we used Swift/T to locate and operate on data in a node-resident Memcached-based filesystem. This enables a powerful data-intensive computing programming model.

Protein science models often run in large ensembles, intermixing parallel computation with dataflow logic. In [10], we used Swift/T to control execution in a molecular dynamics simulator called NAMD. Since NAMD uses Tcl and SWIG at its core, it is able to easily call launch Swift/T TIC, and then expose its internal data structures to Swift/T for control. The Swift/T leaf functions call back into NAMD via the Tcl command uplevel, thus controlling the overarching run. This effort is focused on the Cray Blue Waters.

Swift/T tasks can target GPUs. In [11], we applied dataflow programming to a large number of GPUs on Blue Waters. The GeMTC system was used to expose 86,000 individual GPU warps to Swift/T, which distributed work to them. These tasks were written in CUDA but exposed, as usual, through a Tcl interface.

Swift/T is highly scalable. Peak performance tests conducted on Blue Waters achieved 1.5 billion leaf function tasks per second on 512 thousand compute cores [5]. Other, more complex application patterns have been studied. Swift/T performance has been improved over its initial implementation by improving load balancing at run time as well as via compiler optimizations in STC.

VI. Summary

In this paper, we have presented an overview of how Swift/T works and illustrated its connection to Tcl.

Swift/T may be found at: `http://swift-lang.org/Swift-T`

Acknowledgments

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

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

REFERENCES

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

[2] J. M. Wozniak, T. G. Armstrong, K. C. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster, "Interlanguage parallel scripting for distributed-memory scientific computing," in *Proc. WORKS at SC*, 2015.

[3] D. Beazley, "Automated scientific software scripting with SWIG," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 599–609, 2003.

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

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

[6] 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*, vol. 28, no. 3, 2013.

[7] J. Ozik, N. Collier, and J. M. Wozniak, "Many resident task computing in support of dynamic ensemble computations," in *Proc. MTAGS at SC*, 2015.

[8] J. M. Wozniak, H. Sharma, T. G. Armstrong, M. Wilde, J. D. Almer, and I. Foster, "Big data staging with MPI-IO for interactive X-ray science," in *Proc. Big Data Computing*, 2014.

[9] F. R. Duro, J. G. Blas, F. Isaila, J. Carretero, J. M. Wozniak, and R. Ross, "Exploiting data locality in Swift/T workflows using Hercules," in *Proc. NESUS Workshop*, 2014.

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

[11] 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.