Swift/T: Large-scale Application Composition via Distributed-memory Dataflow Processing

Justin M. Wozniak*‡, Timothy G. Armstrong†, Michael Wilde*‡, Daniel S. Katz‡, Ewing Lusk*‡, Ian T. Foster*†‡
* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA † Dept. of Computer Science, University of Chicago, Chicago, IL, USA ‡ Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

***Abstract*—Many scientific applications are conceptually built up from independent component tasks as a parameter study, optimization, or other search. Large batches of these tasks may be executed on high-end computing systems; however, the coordination of the independent processes, their data, and their data dependencies is a significant scalability challenge. Many problems must be addressed, including load balancing, data distribution, notifications, concurrent programming, and linking to existing codes. In this work, we present Swift/T, a programming language and runtime that enables the rapid development of highly concurrent, task-parallel applications. Swift/T is composed of several enabling technologies to address scalability challenges, offers a high-level optimizing compiler for user programming and debugging, and provides tools for binding user code in C/C++/Fortran into a logical script. In this work, we describe the Swift/T solution and present scaling results from the IBM Blue Gene/P and Blue Gene/Q.**

I. Introduction

Many important application classes that are driving the requirements for extreme-scale systems—branch and bound, stochastic programming, materials by design, uncertainty quantification—can be productively expressed as many-task dataflow programs. High-level dataflow languages are commonly used to solve coarse-grained problems in systems programming and scientific computing. The dataflow programming model of the Swift parallel scripting language can elegantly express, through implicit parallelism, the massive concurrency demanded by these applications while retaining the productivity benefits of a high-level language.

We present here Swift/T, a new dataflow language implementation designed for extreme scalability. Its technical innovations include a distributed dataflow engine that balances program evaluation across massive numbers of nodes using dataflow-driven task execution and a distributed data store for global data access. The Swift/T compiler translates the user script into a fully scalable, decentralized MPI program through the use of enabling libraries. Swift/T further extends the Swift dataflow programming model of external executables with file-based data passing to finer-grained applications using in-memory functions and in-memory data. It directly addresses the intertwined programmability and scalability requirements of systems with massive concurrency with a programming model that may also be attractive and feasible for systems of much lower scale.

We evaluate the performance and programmability of Swift/T for common patterns in distributed computing that make up stress tests of language constructs in Swift/T. Additionally, we evaluate a graph analysis and optimization application. Our tests show that Swift/T can already scale to 128K compute cores with high efficiency. This enables Swift/T to provide a scalable parallel programming model for productively expressing the outer levels of highly-parallel many-task applications.

```
1  int X = 1000, Y = 1000;
2  int A[][];
3  int B[];
4  foreach x in [0:X-1] {
5  foreach y in [0:Y-1] {
6    if (check(x, y)) {
7     A[x][y] = g(f(x),f(y));
8    } else {
9     A[x][y] = 0;
10   }}
11 B[x] = sum(A[x]);
12 }
```

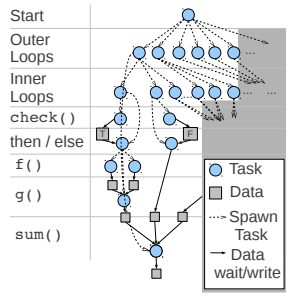

Fig. 1. A simple dataflow application

The benefits of these advances are illustrated by considering the Swift code fragment in Figure 1. In Swift/T, the tasks that the user intends to run may be defined as "leaf tasks", which have a functional syntax. The implicit parallelism of this code generates 1 million concurrent executions of the inner block of expressions, invoking millions of leaf tasks. The Swift/T architecture distributes the evaluation of the outer loop to many processors, each of which can in turn distribute the inner loop to many additional processors. This innovation removes the single-node evaluation bottleneck and enables Swift programs to execute with far greater scalability. The diagram on the right illustrates how evaluation of the entire program – not just the leaf tasks at the leaves of the call graph – can spread through a parallel system and utilize many nodes to rapidly generate massive numbers of concurrent leaf tasks. Thus, this greatly exceeds the scalability of the system beyond that of a task management system based on a single node.

The new Swift/T system is well suited for an emerging class of "many task" applications with the following characteristics:

Non-trivial coordination and data dependencies between tasks, for example with arbitrary directed acyclic graph (DAG) dataflow patterns where dataflow-driven task execution can maximize concurrency. In Swift, the dataflow specification comes not from a static DAG but from the dynamic evaluation of programs written in a concurrent, expressive language.

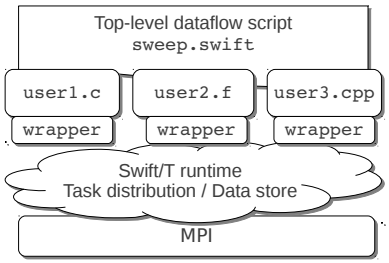

Fig. 2. Overview of Swift/T application model

Irregular or unpredictable computational structure. Many real-world task-parallel problems with irregular structure defeat simple load-balancing approaches because of variable task runtimes, complex dependencies, or irregular data structures. Robust and highly-tuned dynamic load balancing of tasks is essential, but challenging to implement from scratch. Additionally, the underlying load balancer used here offers features not typically available, which are available to Swift/T.

Orchestration of large application codes. Many applications can be expressed naturally by composing complex executable programs or library functions using implicitly parallel dataflow. This development methodology enables rapid development and modification of applications, with performance-critical code expressed in lower-level languages.

As shown in Figure 2, a Swift/T application consists of a Swift/T script and some number of user code "extensions" coded in a native programming language. These consume and produce data in native types, such as integers, floats, pointers to byte data, etc. Swift/T wrappers, which may be automatically generated, allow these extensions to be executed by the runtime. Thus, the user may produce a massively concurrent program without writing MPI directly.

Producing the system involved solving multiple technical challenges. First, the coordination of the Swift/T task/data management cannot be performed on a single node (indeed, as other systems have shown, task management itself at scale must be distributed). Our solution is the development of a *distributed future store* which manages data operations and notification primitives in a fully decentralized manner. Additionally, the high-level Swift language must be compiled to use this unique runtime. Our STC optimizing compiler performs this translation in a novel manner.

The contributions of this work, in which we address the challenges of enabling practical, high-level dataflow programming on large scale, massively multi-node systems, are as follows.

- We describe the characteristics of many-task applications that can benefit from extreme-scale systems (§ II);
- We describe previous related work (§ III);
- We describe the Swift/T programming model and its relation to the runtime system (§ IV, § V);
- We present performance results from common application patterns and a real scientific application that show that the scalability challenges have been overcome (§ VI).

II. Motivating Applications

To demonstrate the value of a highly scalable implementation of the Swift programming language, we present several many-task applications with massive concurrency that have been conveniently expressed as parallel Swift scripts.

A. Scientific collaboration graph analysis

Analysis of graphs of collaboration between scientists is an important first step in enabling the automated discovery of hypotheses or facts [1]. The SciColSim application optimizes a stochastic graph-based model of collaboration by fitting model parameters to actual data mined from publication author lists and other indicators of collaboration.

A simulated annealing algorithm [2] is used to explore the parameter space and improve the fit between model and observation. Pseudo-code for this application is shown in Figure 3. The `evolve` function runs an instance of the stochastic model in order to measure goodness of fit. A production run makes ∼10 million `evolve` calls. Scaling up the application is not trivial, since frequent synchronization is required to aggregate results of parallel tasks and make control-flow decisions, and highly variable task times force frequent load balancing.

```
1 | foreach i in innovation_values {          //  ∼ 20
2 |   foreach r in repeats {                  //  15
3 |     iterate cycle in annealing_cycles {   //  ∼ 100
4 |       iterate p in params {               //  3
5 |         foreach n in reruns {             //  1000
6 |           evolve(...); //  0.1 to 60 seconds
7 |         }}}}}
```

Fig. 3. Loop structure of SciColSim application in Swift. `foreach` indicates a parallel loop and `iterate` a sequential loop

B. Power grid design

Branch and bound algorithms arise in discrete optimization problems: for example, in power grid design optimization using Mixed-Integer Nonlinear Programs [3]. These algorithms recursively subdivide a search space, creating many branches. Branches are pruned once they are of no further interest, for example if they are infeasible or sub-optimal. Detecting sub-optimality may involve shared upper or lower bounds that are updated throughout execution. In parallel implementations, communication is required to share updated bounds between branches. Branch and bound computations are often irregular since the evolution of the search process is unpredictable, so frequent load balancing is required. Task durations depend greatly on the problem and algorithm, but 100s of milliseconds to several minutes [3] is not atypical.

C. Other applications

Ensemble studies involving different methodologies such as uncertainty quantification, parameter estimation, graph pruning, and inverse modeling all require the ability to generate and dispatch tasks in the order of millions to the distributed resources. *Projections of regional crop yields* are computed by running ensemble simulations over data for land cover,

soil, weather, and climate. Investigations are conducted using the Decision Support System for Agrotechnology Transfer (DSSAT) [4], currently run workflows of 480,000 tasks each as composed by Swift. Future DSSAT runs covering additional crops and near-global land area, and intercomparing larger numbers of scenarios will require as much as two orders of magnitude more CPU time per model run and an order of magnitude more models, almost of all which will be structured as many-task computations. *Regional watershed analysis and hydrology* are investigated by the Soil and Water Assessment Tool (SWAT), which analyzes hundreds of thousands of data files via MATLAB scripts on hundreds of cores. This application will utilize tens of thousands of cores and more data in the future. SWAT is a motivator for our work because of the large number of data files. *Biomolecular analysis* via ModFTDock results in a large quantity of available tasks [5], and represents a complex, multi-stage workflow.

III. Background and Related Work

Swift [6] is a parallel scripting language for programming highly concurrent applications in parallel and distributed environments. The language is implicitly parallel with deterministic semantics that aid understanding, debugging, and reproducibility.

The work described here addresses these motivations by solving a fundamental limitation of the previous Swift implementation. Previously, Swift code was evaluated using multiple threads on single centralized node to coordinate external tasks running on (many) additional nodes. While Swift has been used on large clusters, its maximum task dispatch rate is less than 500 tasks per second, and its available memory is limited to that of a single node.

To overcome these limitations, we reimplemented Swift with a new compiler and runtime, together called Swift/T, that allows arbitrary numbers of nodes to cooperate in evaluating a Swift program. Swift/T's innovations are scalable load balancing, distributed data structures, and dataflow-driven concurrent task execution.

The productivity benefits of coordinating high-performance subroutines with scripting has been promoted in the past [7], in particular for HPC applications [8].

The idea of using dataflow to coordinate sequential tasks has been termed macro-dataflow [9]. Data-Driven Tasks [10] supports data-dependent execution of tasks on shared-memory systems within the Habanero Java language. run command-line programs in parallel. CIEL [11] is an execution engine, with a corresponding dataflow scripting language, Skywriting, that also runs tasks as external processes.

Dataflow programming models for HPC applications have been a topic of interest for several groups [12]. Tarragon [13] and DaGuE [14] implement efficient parallel execution of explicit dataflow DAGs of tasks from within an MPI program. TIDeFlow [15] proposes a dynamic dataflow execution model, with execution specified as a (maybe cyclic) graph of dataflow between actors. FOX [16] aims to support dynamic and irregular applications on exascale systems, and uses dataflow graphs for fault tolerance. ParalleX [17] provides a programming model through a C++ library that encompasses globally addressable data and futures, with the ability to launch tasks based on dataflow. Our work is distinguished by its focus on task-parallel applications with moderate task granularity, which may have challenging characteristics such as irregular tasks, unbalanced nested loops, and/or complex data dependencies. We focus on providing an expressive, simple, and robust programming model inspired by scripting languages for top-level application coordination.

The Asynchronous Dynamic Load Balancer (ADLB) [18] is an MPI library for distributing tasks (work units) among worker processes. ADLB is a highly scalable system without a single bottleneck, and has been successfully used by large-scale physics applications. ADLB is a core library used by Swift/T. Scioto [19] is a library for distributed memory dynamic load balancing of tasks, similar to ADLB. Scioto implements work-stealing among all nodes, instead of the server-worker design of ADLB. Scioto's efficiency is impressive, but it does not provide features required for Swift/T such as task priorities, work types, and rank-targeted tasks.

A range of key-value stores exist, such as Dynamo [20], memcached [21], and redis [22]. Their functionality is diverse and varied, but none provide all functionality needed to support Swift/T. Redis is probably the most similar with data structures and publish/subscribe.

IV. Programming Model

We seek to provide a system that allows code written by non-experts to run at extreme scale. This goal might be infeasible in a fully general model for parallel computation. However, we focus on many-task applications, which exhibit simpler coordination patterns but nevertheless can be challenging to scale up in commonly used message-passing programming models. Scalability challenges will only become more daunting on future exascale systems where fault tolerance and power awareness are needed. In the following, we summarize key features of the Swift programming language and the challenges that they pose for the design and implementation of Swift/T.

A. Hierarchical programming

We assume that much performance-critical code will remain in lower level languages such as C, C++, Fortran or even assembly, using threads or MPI for fine-grained parallelism. Dataflow scripting provides a powerful mechanism for coordinating these high-performance components, as it enables fault-tolerance, dynamic load balancing and rapid composition of components to meet new application needs. In Swift, each lower-level component is viewed as a black box with well-defined inputs and outputs. Parallelism is derived by executing these components as parallel tasks.

B. Implicit parallelism

Swift makes parallelism implicit, similarly to other dataflow programming languages such as Sisal [23] and Id [24]. When control enters a code block, any Swift statement in that block can execute concurrently with other statements. This concurrent execution is feasible because of the functional

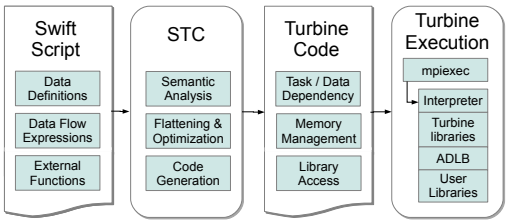

Fig. 4. Schematic of Swift/T usage. Swift scripts are compiled by STC into Turbine code executed by the Turbine runtime

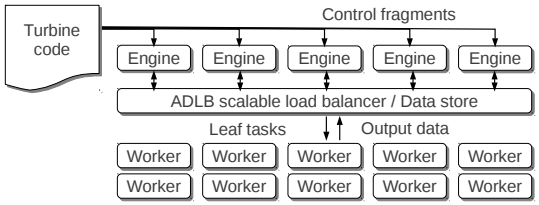

Fig. 5. Architecture of Turbine runtime: Engines evaluate Swift language semantics; workers execute leaf task applications

nature of Swift, where we avoid mutable state and use write-once variables pervasively to schedule execution based on data dependencies. Each operation, down to basic arithmetic, can be realized as an asynchronous task, eligible to be executed anywhere in the distributed-memory computer. This uniformity simplifies language semantics. It is also powerful: any valid Swift expression can appear in any expression context, and thus, for example, an array index can be computed based on the result of a long-running task without any special effort by the programmer. Swift/T ensures that, in all circumstances, work is eligible to run as soon as data dependencies are met, so that all meaningful concurrency present in a user script is retained.

C. Determinism by default

In order for implicit and pervasive parallelism to be manageable we need a simple model for language semantics. It has been argued [25] that parallel languages should have a deterministic sequential interpretation for most language features, with non-determinism only introduced through explicit non-deterministic constructs. All core data types in Swift, *including arrays*, are guaranteed to be deterministic and referentially transparent: that is, querying the state of variable x, or any copy of x with operation f *always* returns the same result, regardless of where $f(x)$ is in the program – even in the case of operations that insert data into an array. Supporting determinism in conjunction with Swift/T's distributed evaluation has been a major challenge.

V. Architecture

The Swift/T architecture consists of two software components: the Swift-Turbine Compiler (STC) and the Turbine scalable runtime. Their usage is shown in Figure 4. STC compiles the user Swift script to the *Turbine code* that is launched as an MPI program with the Turbine runtime system. Turbine was designed to provide efficient support for the Swift data type and dataflow execution model. The Turbine runtime has a library API, with STC generating Turbine code that calls into Turbine through this API.

A. STC compiler architecture

The STC compiler comprises a Swift front end, a series of optimization passes, and a Turbine code generator. The front end of the compiler parses a Swift program, type-checks it, performs dataflow analysis to detect some common errors, such as unassigned variables (important because they can cause deadlocks in Swift), and then generates the Swift-IC intermediate representation. Swift-IC is flatter than Swift code, broken down to individual Turbine operations, which simplifies further analysis and optimization of the program. We have found it to be a useful representation for optimizing distributed dataflow programs.

B. Turbine

The implementation of Turbine has been described previously [26], but we present key features of the system here for completeness.

1) Turbine: Execution model: Turbine enables distributed execution of large numbers of user functions and of control logic used to compose them. Turbine programs are essentially MPI programs that use the ADLB [18] and Turbine libraries. Thus, they can run on any system supporting MPI and can be analyzed using MPI tools such as MPE [27].

Turbine requires the compiler to break user program code into many discrete *fragments*, to enable all work to be load balanced as discrete tasks using ADLB. These fragments are either user-defined leaf tasks, such as external compiled procedures or executables, or *control fragments* for dataflow coordination logic. We refer to an invocation of a fragment, combined with input and output addresses, as a *task*. Turbine engines execute control tasks, while workers execute leaf tasks, as shown in Figure 5.

Turbine tasks are atomically scheduled and execute without pausing or blocking, similar to codelets [28], but with higher granularity. Execution of a Turbine control logic fragment may produce additional control fragments that are redistributed via ADLB.

Turbine must track data dependencies between tasks in order to know when each is eligible to run. Turbine provides a globally-addressable *distributed future store* (see § V-B2, [26]), which drives data-dependent execution and allows typed data operations. A task becomes *ready* once its data dependencies have been satisfied. Small control functions and arithmetic leaf tasks are executed locally to reduce overhead; other tasks are distributed via ADLB. Each task is represented as a binary string containing the fragment to execute, addresses of global input and output data, and serialized scalar values. When a task runs, it fetches its input data, executes, then produces output data, notifying the Turbine data dependency engine, which rapidly releases newly-runnable work.

2) Turbine: Distributed future store: Turbine's distributed future store is used to pass data and to track data dependencies between tasks. The data store was implemented for this work as an additional ADLB service. Primitive data types include 64-bit integers, double-precision floating point numbers, strings, file references, and binary objects (blobs). Turbine provides *write-once variables* for these types, which are used as *futures* [29] for output of asynchronous tasks. A single data structure is provided, the *container*, an associative array. Every Turbine data item starts off in an *open* state, and only once the value is final (i.e., a write-once variable has been written, or a container has had all values inserted), is it switched to the *closed* state. Each data item has a unique 64-bit ID, which is hashed to find its location (an ADLB server), allowing any node in the cluster to access the data. Turbine provides containers that reside on a single data server, but a scalable distributed container abstraction is provided by distributing the contents across many single-node containers by key.

C. Turbine as a Swift runtime

The basic Turbine system just described provided many primitives needed for a Swift runtime, but implementing the full language efficiently and scalably required additional runtime primitives and techniques, described in this section.

To work as part of a scalable Swift runtime, the store must enable two key properties (see § IV): *high concurrency* and *determinism*. Data operations must *tolerate* and *enable* high concurrency by allowing concurrent execution of tasks with shared data and by avoiding extended pauses of tasks. In particular, having tasks suspending waiting for other tasks to perform operations would not interact well with our non-preemptive task dispatcher: deadlock is the worst case, poor utilization more likely. Operations must also provide strong enough guarantees for STC to be capable of generating correct deterministic Turbine code. Our definition of determinism is not completely strict, in that while in the case of a correct program in the deterministic core of Swift, only the order of side-effects such as logging varies, in cases of invalid operations, such as writing a write-once variable twice, an error will be detected but the exact error can vary.

Each execution of a Swift function is realized as the execution of one or more Turbine tasks. Computationally intensive non-Swift functions such as compiled functions or command-line applications execute as Turbine leaf tasks, while control flow in the Swift language is implemented using Turbine control tasks. Turbine tasks never wait for synchronization with, or data from, another task. If, as is often the case, control flow in a Swift function requires multiple waits for data, that Swift function must be compiled to multiple control fragments. We use Turbine's data dependency tracking to launch each fragment at the correct time.

D. Swift/T extension functions

Since Swift/T is a many-task computing language, making external code callable from Swift is crucial. Currently we support calling C and C++ functions from a Swift script, by using SWIG [30] to automatically generate wrappers for C modules, then writing a Swift/T wrapper function to marshal data to and from the Turbine data store. Fortran has been supported by automatically generating a C++ wrapper through FortWrap [31]. Wrapped code can then be made into a Swift/T module and reused in any scripts. For example, we have made modules for the applications evaluated in the performance section and the BLAS library. The BLAS module was developed via CBLAS. As an example, the Swift header definition for `ddot` (dot-product two vectors of doubles) is formulated as follows (note that Swift/T floats are double precision):

```
1  (float z) blas_ddot(int N, float X[], float Y[])
2  {
3    blob x = blob_from_floats(X);
4    blob y = blob_from_floats(Y);
5    z = blas_ddot_blobs(N, x, y);
6  }
7
8  (float z) blas_ddot_blobs(int n, blob x, blob y)
9  "cblas" "0.0.1" "cblas_ddot";
```

Thus, the user may call this function with automatic type conversion or, if the user already has pointers to the C-formatted double arrays, directly with the call to the actual Swift/T leaf tasks, `blas_ddot_blobs()`, which links to the C function `cblas_ddot()`. While our intended user function time granularity is much higher than `ddot` (around 10 seconds), this illustrates how external library routines may be called as leaf tasks from Swift/T programs.

VI. Performance

To demonstrate the practical utility of our implementation, we carried out multiple performance tests. All tests were performed on the IBM Blue Gene/P (BG/P) and Blue Gene/Q (BG/Q) systems at Argonne National Laboratory. BG/Q runs were done on a prototype BG/Q rack. Measurements were made by extracting events from MPE logs.

The BG/P is organized in 4-core nodes. Each 64-bit PowerPC 450 core runs at 850 MHz; each node has 2 GB RAM. The BG/P network is a bidirectional 3D torus; each link has bandwidth 425 MB/s and latency $< 1\mu$s. The BG/Q is organized in 16-core nodes. Each 64-bit PowerPC A2 core runs at 1.6 GHz; each node has 16 GB RAM. The BG/Q network is a bidirection 5D torus; each link has bandwidth 2 GB/s and latency $< 1\mu$s.

We selected four cases for measurement: three application patterns and a non-trivial, real application. The benchmarks measure task management at large scale and raw performance for short tasks. The application case evaluates an application that combines a parameter sweep with iterative optimization, and has longer leaf task run times.

A. Application patterns

We selected three common application patterns to measure Swift/T's ability to manage a large-scale system and rapidly launch tasks on newly released processors. While these benchmarks focus on STC compiler-processed loops, previous results have been reported on hand-coded Turbine loops [26] and STC compiler-generated deep, distributed function call stacks [32].

```
1  foreach i in [0:N-1] {
2    A[i] = f(i);
3  }
```

a) Independent tasks

```
1  foreach i in [0:V-1] {
2    foreach j in [0:V-1] {
3      foreach k in [0:V-1] {
4        foreach m in [0:V-1] {
5          int r = f(i, j, k, m);
6        }}}}
```

b) Nested loops

```
1  A[0][0] = 0;
2  foreach i in [1:N-1] {
3    A[i][0] = 0;
4    A[0][i] = 0;
5  }
6  foreach i in [1:N-1] {
7  foreach j in [1:N-1] {
8    A[i][j] = f(A[i-1][j-1],
9                A[i-1][j],
10               A[i][j-1]);
11 }}
```

c) Wavefront

Fig. 6. Application patterns for performance evaluation

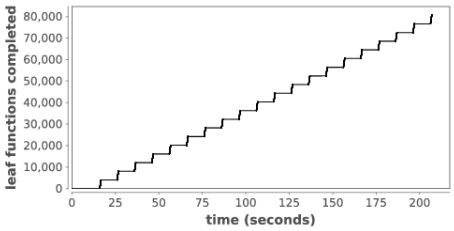

Fig. 7. Task completion events for batch of independent tasks (BG/P)

1) Simple loop pattern: We first evaluate Swift/T's ability to launch and manage a simple bag-of-tasks application, shown in Figure 6 a), at large scale. As shown, this application simply executes leaf task `f()` N times. Each invocation of `f()` emulates a fixed amount of sequential computation time on a Turbine worker.

First, we illustrate the behavior of this script at a high level. A Swift/T run was configured with $P = 4,096$ processes of which 4,032 are workers; 64 are control processes. We set N to 80,640, meaning that each worker executes `f()` 20 times. The computation duration D for `f()` is set to 10 seconds.

Results are shown in Figure 7. Each completion of `f()` increments the cumulative number of leaf tasks completed. As desired, the accumulation over time resembles a step function in which there is a short amount of time between steps, indicating that workers are kept busy.

By measuring the total run time, T, a utilization result for this case may be obtained as

$$\text{utilization } U = \frac{N \times D}{P \times T}. \quad (1)$$

This formula penalizes Swift/T for the use of control processes, which do not perform leaf task work. The utilization for this case is 96.3%.

Second, we scale up both problem size and computer system size in order to evaluate our ability to manage a large number of leaf tasks on many processors. We execute the same script on successively larger problems and for successively larger processor counts. We set $D = 100$ seconds. For each process count P, the number of control processes $C = P \div 64$ and the number of worker processes is $W = P - C$. The number of leaf task invocations is set to $N = W \times 2$; that is, each worker executes the function twice for a total of 200 seconds of work. Utilization is simply:

$$\text{utilization } U = \frac{200}{T}. \quad (2)$$

Results are in Figure 8. At the second largest scale, 65,536 cores, the utilization remains high at 94.57%. At the largest scale, 131,072 cores, the utilization drops to 82.03%. This performance is comparable to that reported for ADLB [18].

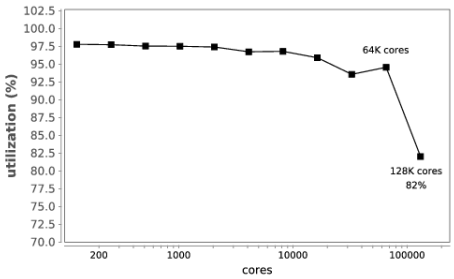

Fig. 8. System utilization for batch of independent tasks (BG/P)

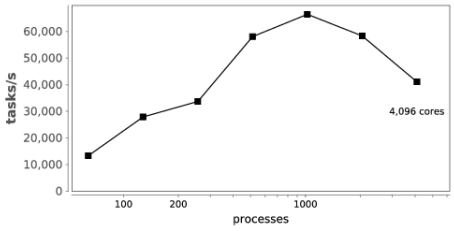

Fig. 9. Task rate result for nested loops pattern (BG/Q)

2) Nested loops pattern: Our second benchmark, like the first, creates a large number of fine-grained tasks, but does so using a quadruple-nested rather than a single `foreach` (see Figure 6). Thus, it tests a different aspect of Swift/T control logic, neglecting the array insertion but storing an output variable. Figure 9 shows the measured task rates. In each case, the number of engines and servers was set to $C = P \div 2$ and script variable V was set such that each server processed at least 2,000 tasks. `f()` simply retrieves the script variables and outputs their sum (no artificial delay).

This test was run on the BG/Q. The plot shows a performance peak at 1,024 cores of 66,448 tasks/s.

3) Wavefront pattern: The wavefront pattern is relevant for many applications [33]. In this framework, a matrix is filled in by results computed by calling a function on three inputs that are adjacent cells in the matrix. In our implementation, the value in each cell is the sum of the entries to its left, top, and top-left. The arithmetic runs on a worker process and runs for essentially 0 seconds. Thus, the test evaluates how fast Swift/T can produce and distribute leaf tasks in this pattern.

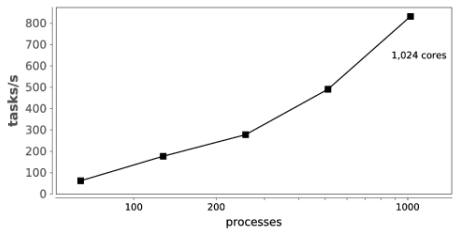

Fig. 10. Task rate result for wavefront pattern (BG/P)

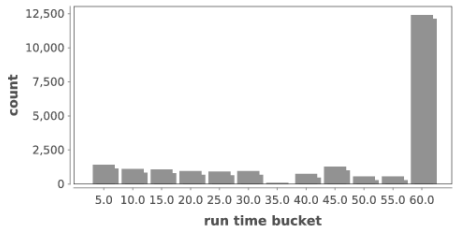

Fig. 11. SciColSim leaf task durations (BG/P)

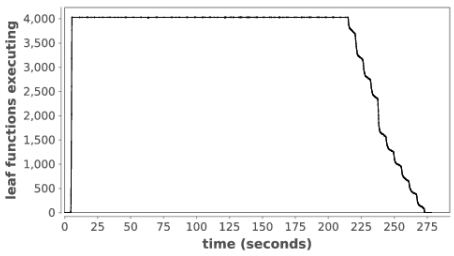

Fig. 12. SciColSim processing load on 4,032 workers (BG/P)

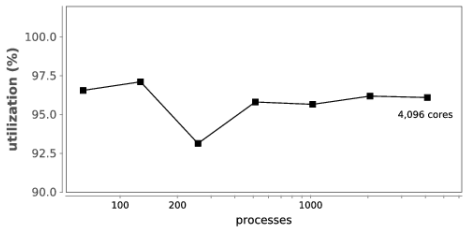

Fig. 13. SciColSim utilization at varying scale (BG/P)

As shown in Figure 10, the wavefront pattern scales on the BG/P well up to 1,024 cores at 832 tasks/s. While real applications are expected to compute more than a simple sum, it demonstrates that applications with relatively short task computation time but complex coordinating logic can benefit from Swift/T.

B. Application: SciColSim

We describe here an application test of the new Swift/T implementation. The application implements the SciColSim simulated-annealing optimization of a social network graph model of scientific collaboration, enabling a multi-core workstation application to be run on the BG/P.

We conducted performance studies for the SciColSim Swift/T application as follows. The SciColSim Swift script (summarized in Figure 3) has 303 lines, replacing a similar amount of OpenMP C++ code. This dataflow script performs simulated annealing. The SciColSim leaf task, `evolve()`, is a graph analysis routine written in C++. The run time distribution for `evolve()` is shown in Figure 11 (this data is extracted from the 4,096-core use case below). Each bar corresponds to the "bucket" of run times that fell below that run time but exceeded the previous bucket. As shown, 55% are between 55 and 60 seconds, 45% are distributed in the range under 55 seconds.

Figure 12 shows leaf task load over time for the 4,096 core case. We see that Swift/T rapidly evaluates the annealing script and launches leaf task execution on all workers within 5.6 seconds; this time includes all job startup. A "long tail" effect is seen as some long-running tasks complete [34].

Figure 13 shows utilization over time at scales up to 4,096 cores. Each system size executes a correspondingly larger SciColSim workload. As shown, each case has utilization 93% or higher (disregarding the long tail effect). This result shows that Swift/T can deliver computing cycles to real application codes as coordinated by complex application scripts.

This tail, problematic on real runs, is solved by using application knowledge to assign higher task priorities to longer-running tasks. Figure 14 shows how using priorities eliminates the tail effect, resulting in an earlier exit. Task priorities are an ADLB feature elegantly exposed in the Swift/T language.

VII. Future Work

We are aware of many potential optimizations to improve Swift/T performance, such as caching, relaxing consistency, and coalescing Turbine operations at compile or run time. Garbage collection is required to support longer running jobs that create more global data. We intend to explore alternative load balancing methods and data-aware scheduling, and expect that advances in this area will yield many-fold improvements to Swift/T's current scalability.

Many system-level features remain to be explored and implemented. The dataflow-driven task-parallel execution model presents opportunities to provide fault tolerance and power awareness at the runtime system level, which we have yet to exploit. We also plan to integrate Swift/T with a MosaStore intermediate file system [35] deployed on the compute nodes to support efficient workflow-aware file access [36] and cache-resident program executables as tasks.

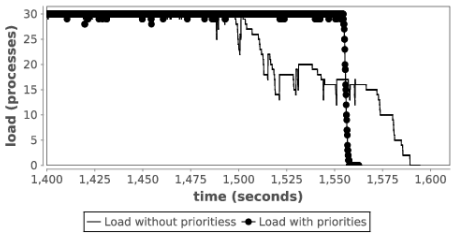

Fig. 14. SciColSim utilization trace on 30 workers for 32 separate annealing processes, each 5 cycles long, with 5-way parallelism in each process

VIII. Conclusion

The novel contribution of this research is the design, implementation, validation, and evaluation of Swift/T, a completely new implementation of the implicitly-parallel Swift language for high-performance computers. This work has yielded a practical dataflow-based programming model for productively implementing the upper-level logic of complex many-task applications on emerging extreme-scale platforms. The principal innovation of Swift/T is its implementation of highly-distributed execution for parallel dataflow-based semantics through the integration of a scalable task distribution model, a distributed data store, and requisite compilation techniques.

Performance results show good utilization for realistic workloads at task rates that far exceed previous systems. Several application classes which can benefit from this programming model on extreme-scale systems were discussed, and specific examples were provided and cited. Swift/T has been used to develop and execute highly scalable real-world applications through parallel composition of C/C++/Fortran functions.

Acknowledgments

This research is supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357, FWP-57810. Computing resources were provided by the Argonne Leadership Computing Facility. This material was based on work (by Katz) supported by the National Science Foundation, while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Evans and A. Rzhetsky, "Machine science," *Science*, vol. 329, no. 5990, pp. 399–400, 2010.

[2] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[3] A. Mahajan, S. Leyffer, and C. Kirches, "Solving mixed-integer nonlinear programs by QP-Diving," Mar. 2012, preprint ANL/MCS-P2071-0312.

[4] J. W. Jones, G. Hoogenboom, P. Wilkens, C. Porter, and G. Tsuji, Eds., *Decision Support System for Agrotechnology Transfer Version 4.0: Crop Model Documentation*. University of Hawaii, 2003.

[5] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: uniform resource provisioning and access for scientific computing on clouds and grids," in *Proc. Utility and Cloud Computing*, 2011.

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

[7] J. K. Ousterhout, "Scripting: higher level programming for the 21st century," *Computer*, vol. 31, no. 3, pp. 23 –30, Mar. 1998.

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

[9] V. Sarkar and J. Hennessy, "Partitioning parallel programs for macro-dataflow," in *LFP'86*. New York, NY, USA: ACM, 1986, pp. 202–211.

[10] S. Tasirlar and V. Sarkar, "Data-Driven Tasks and their implementation," in *Int'l Conf. on Parallel Processing (ICPP) 2009*, Sep. 2011, pp. 652–661.

[11] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: a universal execution engine for distributed data-flow computing," in *Proc. NSDI 2011*, 2011.

[12] S. Evripidou, G. Gao, J.-L. Gaudiot, and V. Sarkar, Eds., *1st Workshop on Data-Flow Execution Models for Extreme Scale Computing: DFM'11*. IEEE Computer Society, Oct. 2011.

[13] P. Cicotti, "Tarragon: a programming model for latency-hiding scientific computation," Ph.D. dissertation, U. California, San Diego, 2011.

[14] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," U. Tennessee, Tech. Rep. ICL-UT-10-01, 2012Apr. 2010.

[15] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao, "TIDeFlow: The time iterated dependency flow execution model," in *1st Workshop Data-Flow Execution Models for Extreme Scale Computing: DFM'11*, Oct. 2011, pp. 1–9.

[16] R. G. Minnich, C. L. Janssen, S. Krishnamoorthy, A. Marquez, M. Gokhale, P. Sadayappan, E. Van Hensbergen, J. McKie, and J. Appavoo, "Fault oblivious eXascale whitepaper," in *Proc. 1st Int'l Workshop Runtime and Operating Systems for Supercomputers: ROSS'11*. New York: ACM, 2011, pp. 17–24.

[17] H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An advanced parallel execution model for scaling-impaired applications," in *Int'l Conf. Parallel Processing Workshops (ICPPW) 2009*, Sep. 2009, pp. 394–401.

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

[19] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan, "Scioto: A framework for global-view task parallelism," *Int'l Conf. on Parallel Processing*, pp. 586–593, 2008.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[21] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, pp. 5–, Aug. 2004.

[22] S. Sanfilippo and P. Noordhuis, "Redis," http://redis.io/.

[23] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the Sisal language project," *J. Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349 – 366, 1990.

[24] K. R. Traub, "A compiler for the MIT tagged-token dataflow architecture," Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep., 1986.

[25] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Workshop Hot Topics in Parallelism: HotPar'09*. Berkeley, CA: USENIX Association, 2009.

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

[27] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Par. Comp.*, vol. 22, no. 6, pp. 789–828, 1996.

[28] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "codelet" program execution model for exascale machines," in *Workshop Adaptive Self-Tuning Comp. Syst. for the Exaflop Era, 2011*. New York: ACM, 2011, pp. 64–69.

[29] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proc. of 1977 Symp. on Artificial Intelligence and Programming Languages*. New York: ACM, 1977, pp. 55–59.

[30] D. M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*. Berkeley, CA, USA: USENIX Association, 1996.

[31] J. McFarland, "FortWrap web site," http://fortwrap.sourceforge.net.

[32] T. G. Armstrong, J. M. Wozniak, M. Wilde, K. Maheshwari, D. S. Katz, M. Ripeanu, E. L. Lusk, and I. T. Foster, "ExM: High level dataflow programming for extreme-scale systems," in *Workshop Hot Topics in Parallelism: HotPar'12*, 2012.

[33] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain, "Harnessing parallelism in multicore clusters with the All-Pairs, Wavefront, and Makeflow abstractions," *Cluster Computing*, vol. 13, pp. 24–256, 2010.

[34] T. G. Armstrong, Z. Zhang, D. S. Katz, M. Wilde, and I. T. Foster, "Scheduling many-task workloads on supercomputers: Dealing with trailing tasks," in *Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.

[35] S. Al-Kiswany, A. Gharaibeh, and M. Ripeanu, "The case for a versatile storage system," *SIGOPS Oper. Sys. Rev.*, vol. 44, no. 1, pp. 10–14, Mar. 2010.

[36] E. Vairavanathan, S. Al-Kiswany, L. Costa, M. Ripeanu, Z. Zhang, D. Katz, and M. Wilde, "Workflow-aware storage system: An opportunity study," in *Proc. CCGrid*, 2012.