Swift/T: Scalable Data Flow Programming for Many-Task Applications

Authors removed for submission

Abstract

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 data flow programs. The data flow 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.

However, the centralized single-node evaluation model of the present Swift implementation limits scalability. Overcoming this limitation, while important, is also difficult, as seen by the absence of any massively-scalable data flow languages in current use. The primary challenge is the efficient integration of highly distributed task load balancing with global access to shared data.

We present here Swift/T, a new data flow language implementation designed for extreme scalability. Its technical innovations include a distributed data flow engine that balances program evaluation across massive numbers of nodes using data-flow-driven task execution and a distributed data store for global data access. Swift/T extends the Swift data flow programming model of external executables with file-based data passing to finer-grained applications using in-memory functions and in-memory data.

We evaluate the performance and programmability of Swift/T for a collaboration graph analysis and optimization application, a branch-and-bound game solver, and synthetic stress tests of languages by design, uncertainty quantification—can be expressed naturally by composing complex executable programs.

1. Introduction

Swift [36] 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 Swift’s 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 from scratch 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 data flow-driven concurrent task execution. The benefits of these advances are illustrated by considering the Swift code fragment in Figure 1.

Figure 1. A simple data flow application.

The implicit parallelism of this code generates 1 million concurrent executions of the inner block of expressions, invoking as many as 4M function calls (3M within conditional logic). Previously, the single-node Swift engine would perform the work of sending these leaf function tasks to distributed CPUs at <500 tasks/sec. The new Swift/T architecture, in contrast, can distribute the evaluation of the outer loop to many CPUs, each of which can in turn distribute the inner loop to many additional CPUs. 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 external tasks at the leaves of the call graph – can utilize many nodes to rapidly generate massive numbers of leaf tasks.

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) data flow patterns where data flow-driven task execution can maximize concurrency. In Swift, the data flow specification comes not from a static DAG but from the dynamic evaluation of programs written in a highly concurrent, expressive language.

- 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.

- Orchestration of large application code. Many applications can be expressed naturally by composing complex executable pro-

[Copyright notice will appear here once ‘preprint’ option is removed.]
grams or library functions using implicitly parallel data flow. This development methodology enables rapid development and modification of applications, with performance-critical code expressed in lower-level languages.

The contributions of this work, in which we address the challenges of enabling practical, high-level data flow 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.
- We describe a scalable Swift implementation that uses distributed parallel execution of a Swift program to coordinate execution of application tasks.
- We detail key implementation challenges and techniques for deterministic script evaluation with high concurrency.
- We describe the STC compiler, which compiles Swift to low-level intermediate code, and show compiler optimizations that markedly reduce coordination overhead.
- We present performance results that show that the scalability challenges have been overcome.

2. 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 can be (and in many cases have been) conveniently expressed as parallel Swift scripts.

2.1 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 [15]. 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 [23] is used to explore the parameter space and improve the fit between model and observation. Pseudo-code for this application is shown in Figure 2. 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.

```
foreach i in innovation_values { // ~ 20
    foreach r in repeats { // 15
        iterate cycle in annealing_cycles { // ~ 100
            iterate p in params { // 3
                foreach n in reruns { // 1000
                    evolve(...); // 0.1 to 60 seconds
                }
            }
        }
    }
}
```

Figure 2. Loop structure of SciColSim application in Swift. foreach indicates a parallel loop and iterate a sequential loop.

2.2 Branch and bound for power grid design

Branch and bound algorithms arise in discrete optimization problems: for example, in power grid design optimization using Mixed-Integer Nonlinear Programs [26]. 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

2.3 Socio/climate/crop modeling for sustainable food supply

Several international projects [1–3], are collaborating to model the interactions of climate and socio-economic changes on the supply and demand of agricultural commodities, land-use and land-cover, and global food sustainability. The Decision Support System for Agrotechnology Transfer (DSSAT) [21] analyzes the effects of climate change on agricultural production.

Projections of regional crop yields are computed by running ensemble simulations over data for land cover, soil, weather, and climate. Figure 3 shows a DSSAT simulation in Swift, running four campaigns of 120,000 tasks each. 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.

2.4 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. 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 [20], and represents a complex, multi-stage workflow.

2.5 Summary

Table 1 summarizes required task rates for a full utilization of 10^6 cores at a stable state. Excluding the ramp up and ramp down stage, a steady flow of tasks per second is determined by a division of number of cores by task duration.

3. Background and Related Work

The productivity benefits of coordinating high-performance subroutines with scripting has been promoted in the past [30], in particular for HPC applications [9].
The idea of using data flow to coordinate sequential tasks has been termed macro-data flow [32]. Data-Driven Tasks [33] supports data-dependent execution of tasks on shared-memory systems within the Habanero Java language. run command-line programs in parallel. CIEL [28] is an execution engine, with a corresponding data flow scripting language, Skywriting, that also runs tasks as external processes.

Data flow programming models for HPC applications have been a topic of interest for several groups [16]. Tarragon [12] and DaGuE [11] implement efficient parallel execution of explicit data flow DAGs of tasks from within an MPI program. TIDEFlow [29] proposes a dynamic data flow execution model, with execution specified as a (maybe cyclic) graph of data flow between actors. FOX [27] aims to support dynamic and irregular applications on exascale systems, and uses data flow graphs for fault tolerance. ParalleX [22] provides a programming model through a C++ library that encompasses globally addressable data and futures, with the ability to launch tasks based on data flow. 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) [25] 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 [14] 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 [13], memcached [18], and redis [31]. 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.

### 4. 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.

#### 4.1 Hierarchical programming

We assume that much performance-critical code will remain in lower level languages such as C, Fortran or even assembly, using threads or MPI for fine-grained parallelism. data flow 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.

#### 4.2 Implicit parallelism

Swift makes parallelism implicit, similarly to other data flow programming languages such as Sisal [17] and Id [34]. 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 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. In order to make this model perform acceptably well, our compiler often coalesces fine-grained tasks or replaces them with serial code if no gains are expected from parallelism (see §5.5).

#### 4.3 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 [10] that parallel languages should have a deterministic sequential interpretation for most language features, with nondeterminism 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.

### 5. Architecture

The Swift/T architecture consists of two software components: the Swift-Turbine Compiler (STC) and the Turbine scalable runtime.

#### Table 1. Estimated application scales for 1M-core systems.

<table>
<thead>
<tr>
<th>Application</th>
<th>Measured Tasks</th>
<th>Task Dur.</th>
<th>Required Tasks</th>
<th>Task Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power-grid Distribution</td>
<td>10,000</td>
<td>13 s</td>
<td>10⁹</td>
<td>6.6 x 10⁷/s</td>
</tr>
<tr>
<td>DSSAF</td>
<td>500,000</td>
<td>12 s</td>
<td>10⁸</td>
<td>8.3 x 10⁷/s</td>
</tr>
<tr>
<td>SciFiOsim</td>
<td>10,800,000</td>
<td>10 s</td>
<td>10⁸</td>
<td>10⁹/s</td>
</tr>
<tr>
<td>SWAT</td>
<td>2,200</td>
<td>120 s</td>
<td>10⁸</td>
<td>8.3 x 10⁷/s</td>
</tr>
<tr>
<td>ModFTDock: stages</td>
<td>1,200,000</td>
<td>1,000 s</td>
<td>10⁶</td>
<td>10⁹/s</td>
</tr>
<tr>
<td>modmerge</td>
<td>12,000</td>
<td>5 s</td>
<td>10⁵</td>
<td>2 x 10⁷/s</td>
</tr>
<tr>
<td>score</td>
<td>12,000</td>
<td>6,000 s</td>
<td>10⁵</td>
<td>166/s</td>
</tr>
</tbody>
</table>

#### Figure 4. Schematic of Swift/T usage. Swift scripts are compiled by STC into Turbine code executed by the Turbine 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 data flow execution model. The Turbine runtime has a library API, with STC generating Turbine code that calls into Turbine through this API.

5.1 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 data flow 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 (see §5.4). 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 data flow programs.

The optimization phase of the compiler works by iteratively rewriting the Swift-IC representation, cycling through multiple optimization passes. After optimization, the code generator produces Turbine code ready to run as an MPI job.

5.2 Turbine

The implementation of Turbine has been described previously [37], but we present key features of the system here for completeness.

5.2.1 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 [25] and Turbine libraries. Thus, they can run on any system supporting MPI and can be analyzed using MPI tools such as MPE [19].

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 functions, such as external compiled procedures or executables, or control fragments for data flow 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 functions, as shown in Figure 5.

Turbine tasks are atomically scheduled and execute without pausing or blocking, similar to codelets [38], 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 §5.2.2, [37]), 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 functions 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.

5.2.2 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 [8] 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.

5.3 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 §4): 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.

5.3.1 Mapping Swift functions onto Turbine tasks

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.

5.3.2 Commutative non-blocking operations

The primary technique used to achieve determinism without sacrificing concurrency is commutative non-blocking operations for the future store. Commutativity means that, given a set of operations with the property applied to an item in the data store, then regardless of the time order in which the operations occur, the outcome is the same [24]. If these operations are used correctly, this made it possible for our compiler to generate Turbine code that was correct independent of task scheduling, which is crucial when concurrently executing tasks share data. Non-blocking operations are
immediately
L[1] and L[0] done immediately
L[3] done

Figure 6. Two equivalent sequences of commutative container operations: lookups (L), inserts (I), increments (Incr), and decrements (Decr). Gray cells indicate the container is closed.

necessary because they can be used without stopping the progress of task execution. Turbine already supported some operations with the required properties, but many were added during Swift/T implementation so that all Swift data types had the requisite operations.

A Turbine write-once variable supports two commutative operations: subscribe and store. Subscribe guarantees that once a variable is written, a notification will be sent to a specified node. Store assigns a value to the variable, which is used for Swift arrays and structs. The commutative operations:

- lookups (L),
- inserts (I),
- increments (Incr), and
decrements (Decr).

Regardless of the order of the store and subscribes, the outcome is always the same. If there are multiple stores, an error is always raised; the exact error depends on task scheduling. Neither operation blocks task execution, requiring at most one message round-trip.

A further operation, rule, which executes a task once a set of variables is closed and commutes with the two prior operations, is implemented using subscribe. Deterministic data flow operations can then be implemented with rule. For example, a plus operation uses rule to wait until its two operands are written, then runs a task that retrieves the values of the operands, sums them and stores the result in an output variable. The implementation of this operation also requires the non-commutative load operation to retrieve operand values.

### 5.3.3 Commutative operations for associative arrays

Turbine has an associative array data type called a container that is used for Swift arrays and structs. The commutative operations:

- insert and lookup require both a container and a scalar key as arguments.
- Insert immediately inserts the address of an existing Turbine variable, failing if there was a previous insert for that key.
- Lookup returns a future for the address of the Turbine variable at that key, which we term a reference since it adds another level of indirection compared with a regular future. This approach allows lookups that occur before an insert to eventually result in the correct address.

We require lookups to fail if the key is never written, and that the size and enumerate operations on containers be deterministic. Thus, we need to be able to determine when a container is closed. To this end, we maintain a writers counter for each container, which is incremented or decremented to track the number of active concurrent tasks that might insert data. A container is considered closed once its writers counter drops to zero, with any subsequent inserts raising an error. Our compiler handles incrementing and decrementing, so this counting is transparent to Swift/T programmers. Figure 6 illustrates two sequences of commutative operations on a container.

Additional commutative operations are supported for nested containers, which Swift supports in order to represent ragged multidimensional arrays.

### 5.3.4 Limited non-determinism

Some patterns are difficult to express efficiently with write-once variables, for example branch pruning in branch and bound algorithms (e.g., see §6.2.1) and shared counters. Updatable variables can better support such patterns. An updatable variable is initialized to a fixed number, and can then be updated with one of several commutative update operations: increment, max, min or scale. The value retrieved by each read will not be deterministic, but the commutativity property makes the non-determinism more predictable than a variable supporting arbitrary mutation.

### 5.3.5 Unsafe operations

The commutative operations in many cases require additional indirection, such as references, which add some overhead. We implemented multiple, more efficient, unsafe variants for common operations that avoid creating references or subscribing to argument variables but do not provide commutativity. Compiler analysis (see §5.5) is used to find opportunities to replace safe with unsafe operations, improving performance without affecting correctness.

### 5.4 Swift-IC intermediate code representation

We describe briefly the structure of Swift-IC, a simple example of which is shown in Figure 7. The basic unit of code is a block: a series of abstract instructions such as Turbine operations and asynchronous calls to other functions. alloc statements at the top of blocks allocate variables, and control flow constructs are implemented with continuations that execute synchronously or asynchronously. Variables declared in a block are accessible to nested continuations, although sometimes the pass in annotation must be used to explicitly pass variables, avoiding passing unneeded data between tasks. Swift-IC requires two other annotations: #deepopen helps with container writer count bookkeeping, and #awaiton suspends entry into a function until variables are closed.

Each Swift primitive type has three storage options in Swift-IC: local (a temporary stack variable in a program fragment), future, and reference. In Swift-IC, we denote these three storage options by prefixing a type name with $ prefix, no prefix, and *, respectively. For example, with integers the respective types are $int, int, and *int. The options have progressively more overhead, so we use the value version where possible, and use references for Swift array lookup semantics (see §5.3.2).

Continuations come in several flavors. Wait continuations encapsulate asynchronous blocks that run after a set of Turbine variables is closed. If and switch continuations run synchronously after the parent block, branching based on the value of a local variable. To branch on a future, we wrap wait around an if or switch, with a load instruction used to fetch the value of the variable.

Loops come in three varieties: parallel iteration over the contents of an array uses foreach, parallel iteration over an arithmetic series uses range, and sequential iteration uses loop. Each sequential loop iteration can pass data to the next, remapping names to different immutable variables each iteration. A sequential ordering of iterations is therefore necessary, but iterations can overlap when data dependencies permit.

### 5.5 Compiler optimization

The front end of the compiler is simplified by using general-purpose but inefficient translations of Swift constructs to produce correct but inefficient Swift-IC. By focusing on optimization of Swift-IC, we have been able to implement more general optimizations with less effort. For example, STC transforms the initial unoptimized Swift-IC in the middle of Figure 7 to the leaner code at the bottom of the figure, in which constant expressions have been evaluated, unneeded global futures have been eliminated or converted to local variables, and heavyweight asynchronous data flow arithmetic replaced with inline arithmetic (the local_op calls). One wait was eliminated and the other was moved to the waiton list, ensuring fib is not entered until n is closed and thereby avoid a separate task creation for the wait. In combination, these changes reduce the coordination overhead of fib several-fold.
Figure 7. Naïve recursive algorithm for nth Fibonacci number, in which $f(n - 1)$ and $f(n - 2)$ are calculated by asynchronous parallel tasks. Swift-IC outputs precede inputs.

Table 2. Turbine operation counts in SciColSim run by STC optimization level. Each row includes prior optimizations.

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Rule</th>
<th>Store</th>
<th>Load</th>
<th>Subscribe</th>
<th>Insert</th>
<th>Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>52422</td>
<td>42464</td>
<td>78470</td>
<td>139005</td>
<td>5871</td>
<td>11445</td>
</tr>
<tr>
<td>+ Clp + DCElim</td>
<td>52422</td>
<td>41629</td>
<td>77434</td>
<td>112857</td>
<td>5871</td>
<td>11445</td>
</tr>
<tr>
<td>+ Const share</td>
<td>52422</td>
<td>30174</td>
<td>77454</td>
<td>112852</td>
<td>5871</td>
<td>11445</td>
</tr>
<tr>
<td>+ Fwd data-flow</td>
<td>4114</td>
<td>4681</td>
<td>12272</td>
<td>15437</td>
<td>5871</td>
<td>10645</td>
</tr>
<tr>
<td>+ Unroll loops</td>
<td>4014</td>
<td>4643</td>
<td>12111</td>
<td>15213</td>
<td>5871</td>
<td>10595</td>
</tr>
</tbody>
</table>

The goals of STC’s optimizer are somewhat different from those of a traditional compiler. The optimizer must, in rough order of importance, attempt to:

- Preserve parallelism by avoiding the introduction of artificial dependencies between leaf tasks.
- Optimize for scalability of execution, for example by partitioning parallel loops in ways that help load balancing.
- Reduce Turbine overhead, particularly communication, by eliminating Turbine data operations and task creations.
- Optimize for sequential speed of generated control code.

We have applied standard compiler optimizations to the problem of reducing Turbine operations. To demonstrate their effectiveness, we compiled and ran the SciColSim application with different optimizations applied. As a proxy for the Swift/T runtime overhead, we count the number of times key Turbine operations that involve interprocess communication are performed. Figure 7 shows the incremental improvements obtained on a short SciColSim run. The numbers of operations performed in each category were reduced significantly, some by an order of magnitude. The optimizations used are all adaptations of standard techniques described in the compiler literature [4], but applied here to allow Swift’s high-level, uniform abstractions to be compiled to efficient Turbine code. We describe them briefly in the following.

Constant folding and propagation (Clp) guarantees that there is no performance penalty for using constant variables or expressions, compared to hardcoded numbers. Constant sharing avoids repeated allocation and initialization of variables holding constant data. Dead code elimination (DCElim) identifies and removes unneeded variables and instructions.

Forward data-flow analysis, the optimization pass that has the most dramatic impact, tracks which expressions are available and which variables are closed at each point in the intermediate code. This data is exploited to implement the remaining three optimizations. Common subexpression elimination eliminates many redundant computations, and also performs intra-task and inter-task caching of array and structure lookups, reducing data store load. Wait elimination is performed if variables are detected to be closed, reducing overhead from additional asynchronous tasks and tracking of data dependencies. Strength reduction substitutes general-purpose operations with specialized ones, for example, inline arithmetic in place of data flow arithmetic, unsafe data operations instead of commutative ones, or local variables in place of futures. Strength reduction is commonly applied to array indices, which are usually either constants or simple expressions based on loop counters. In both cases, strength reduction swaps indices from futures to local variables, allowing the expressive benefits of arbitrary array indices in Swift/T without overhead when expressivity is unnecessary.

The wait block is a new Swift construct that waits for n argument variables to be closed then runs a block of code. User-inserted wait statements can enable strength reduction and other optimization, since variables are closed in the block. Often a cascade of
strength reduction occurs, with all intermediate results stored in local variables. In the future we would like to automatically identify opportunities to do this.

5.6 Performance hints through code annotations

For code transformations that can sometimes impair performance, we provide a system of annotations that give performance hints without changing program semantics. The @sync annotation applied to a foreach loop or function definition avoids spawning an asynchronous control task per loop iteration or function call. The @unroll=k annotation unrolls foreach loops, allowing optimizations across loop iterations. The @splitdegree annotation allows tunable loop-splitting, generating code to repeatedly perform n-way splits of the loop iterations until chunks of n iterations are obtained, allowing quicker ramp-up of large loops by distributing control flow across Turbine engines.

5.7 Static versus runtime optimization

Runtime optimizations such as caching are complementary to static optimization. The data in Figure 8 demonstrates a highly skewed access distribution for a typical application: 61% of the accesses are to 11.4% of the variables (500/4378), meaning a cache could achieve a high hit rate. Since most data is write-once, aggressive caching of most data operations is possible without cache consistency concerns.

![Figure 8. Distribution of reads to global write-once variables in a SciColSim run when using the full suite of optimizations.](image)

5.8 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 to automatically generate wrappers for C modules, then calling a Swift/T wrapper function to marshal data to and from the Turbine data store. 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, Sudoku and SciColSim.

6. 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 systems at Argonne National Laboratory: runs of 4,096 cores or less were done on Surveyor; larger runs were done on Intrepid. Measurements were made by extracting events from MPE logs.

The Blue Gene/P (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 < 1μs.

We selected four cases for measurement: two synthetic benchmarks and two applications. The benchmarks measure task management at large scale and raw performance for short tasks. The application cases measure an application in the recursive search pattern with short leaf function run times and an application that combines a parameter sweep with iterative optimization, and has longer leaf function run times.

6.1 Benchmarks

We use two benchmarks 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 [37] and STC compiler-generated deep, distributed function call stacks [7].

6.1.1 Naturally parallel pattern

We first evaluate Swift/T’s ability to launch and manage a simple bag-of-tasks application, shown in Figure 9 a), at large scale. As shown, this application simply executes leaf function 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 10. Each completion of f() increments the cumulative number of leaf functions 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

\[
\text{utilization } U = \frac{N \times D}{P \times T}. \tag{1}
\]

This formula penalizes Swift/T for the use of control processes, which do not perform leaf function 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 functions 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 function 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:

\[
utilization \ U = \frac{200}{T}.
\]

Results are in Figure 11. 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 elsewhere [25].

### 6.1.2 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 \texttt{for}\texttt{each} (see Figure 9). Thus, it tests a different aspect of Swift/T control logic, neglecting the array insertion but storing an output variable. Figure 12 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. \texttt{f()} simply retrieves the script variables and outputs their sum (no artificial delay).

The plot shows a performance peak at 1.024 cores. For the 4,096 core case, MPE profiling information reveals the time spent in three performance-critical calls as follows:

\texttt{Task put: 1.428s; Task get: 525,706s; Data store: 1.064s}

These measurements partition latency between data flow processing and task distribution. (The data store operation includes notification processing for data flow progress.) By experimentally removing the ADLB latency, this extreme test case achieves 49,754 task/s. We conclude that task distribution is the bottleneck here, and expect to focus much future effort on its improvement.

### 6.2 Applications

We describe here two application tests of the new Swift/T implementation. The first test explores using Swift data flow to express

```swift
|
| (board next[]) solver(updateable_float done, blob board, boolean bfs, int quota) |
| "sudoku" "0.0" "sudoku_step" |
| main() |
| \{ |
| updateable_float done = 0.0; |
| int n = turbine_workers(); |
| blob b1 = parse_board(argv("board")); |
| board split[] = solver(done, b1, true, n); |
| @sync foreach b3 in split2 |
| int bsize = toint(argv("bsize")); |
| board split[] = solver(done, b2.board, true, n); |
| \} |
| \} |
| sudoku_solve(updateable_float done, blob candidate, int filled) |
| \{ |
| if (done == 0.0) |
| \{ // Terminate once solved |
| int q = toint(argv("dfs_quota")); |
| board split[] = solve(done, b2.board, true, n); |
| \} |
| \} |
| \} |
```

branch-and-bound programming techniques (using a game evaluator as a test laboratory). The second test implements the SciColSim simulated-annealing optimization of a social network graph model of scientific collaboration, enabling a multi-core workstation application to be scaled to the Blue Gene/P.

### 6.2.1 Branch-and-bound

To explore approaches to implementing branch and bound algorithms (see §2.2) in Swift, we use a simple problem of this class: solving a Sudoku board.

We first implemented a serial solver in C, using bitwise operations and compiler intrinsics to implement simple deduction efficiently. The solver attempts deductions, and once it cannot make any more progress, divides the search space on values of the most constrained cell. The solver can perform both breadth first and depth first search (BFS and DFS).

To parallelize the code we created Swift/T bindings from the solver’s C header file with SWIG [9]. The Swift script, shown in Figure 13, ramps up execution by using BFS to rapidly divide the search space, then switches to DFS. During DFS, the C solver code periodically yields control, returning a list of candidates, allowing work to be shared. Priority annotations (\texttt{#p}) prioritize execution of subproblems with more filled squares, keeping the number of concurrent tasks manageable. Task runtimes are highly skewed, because many tasks quickly exhaust all viable options in their search space. Implementing this application in a hierarchical programming model is a simple and powerful way to achieve load balancing of efficient serial subroutines with relatively little effort.

To illustrate the performance characteristics of this application, we first describe results from a single workstation run on a 4-core Intel i7-2760 on an easily-solved 100x100 board. As shown in Figure 14, varying the DFS level has a significant impact on program

![Figure 11. System utilization for batch of independent tasks](image1)

![Figure 12. Task rate result for nested loops pattern](image2)

![Figure 13. Swift script for Sudoku problem with only type declarations and module imports omitted. solve is the Swift leaf function bound to the C solver code. Each board state is represented as a struct with the number of filled cells and the board data. The Swift script, shown in Figure 13, ramps up execution by using two layers of BFS, then sudoku\_solve performs recursive search. Program arguments are accessed using argv.](image3)
behavior. As the DFS level is increased, the number of leaf function calls is reduced proportionately. The wall time performance is recorded as the total time to solution. As shown, there is a sweet spot at DFS=128 for a minimum run time of 4.96s.

We tested the effectiveness of the parallelization on a harder problem on a small network of quad-core computers (see Figure 15). At 32 cores, the problem is solved in around 8s, by which point time is dominated by the initial ramp-up, and the time taken for branches to be cancelled and ADLB to shut down. Average task times were around 250ms. The utilization trace shows that high steady state utilization is achievable on longer-running problems.

6.2.2 SciColSim

We conducted performance studies for the SciColSim Swift/T application as follows. The SciColSim Swift script, not shown here, has 303 lines, replacing a similar amount of OpenMP C++ code. This data flow script performs simulated annealing. The SciColSim leaf function, `evolve()`, is a graph analysis routine written in C++. The run time distribution for `evolve()` is shown in Figure 16 (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 17 shows leaf function load over time for the 4,096 core case. We see that Swift/T rapidly evaluates the annealing script and launches leaf function 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 [6].

Figure 18 shows utilization over time at scales up to 4,096 cores. Each system size executes a correspondingly larger SciColSim workload. Utilization is computed according to Equation 1. 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 19 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.

7. 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 data flow-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 MosaicStore intermediate file system [5] deployed on the compute nodes to support efficient workflow-aware file access [35] and cache-resident program executables as tasks.

8. 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. This work has yielded a practical data flow-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 data flow-based semantics through the integration of a scalable task distribution model, a distributed data store, and the compilation techniques needed to translate Swift into this new runtime environment. Traditional compiler optimizations allowed marked reduction in runtime synchronization overhead. Performance results show good utilization for realistic workloads up to 131,072 cores on a Blue Gene/P. 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 existing C/C++ functions.

References