Fundamenta Informaticae 128 (2013) 337–366
DOI 10.3233/FI-2013-949
IOS Press

Turbine: A Distributed-memory Dataflow Engine for High Performance Many-task Applications

Justin M. Wozniak[*]
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL USA
wozniak@mcs.anl.gov

Ketan Maheshwari
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL USA
ketan@mcs.anl.gov

Daniel S. Katz
Computation Institute, University of Chicago &
Argonne National Laboratory, Chicago, IL USA
d.katz@ieee.org

Ian T. Foster
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL USA
foster@mcs.anl.gov

Timothy G. Armstrong
Computer Science Department
University of Chicago, Chicago, IL USA
tga@uchicago.edu

Ewing L. Lusk
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL USA
lusk@mcs.anl.gov

Michael Wilde
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL USA
wilde@mcs.anl.gov

Abstract. Efficiently utilizing the rapidly increasing concurrency of multi-petaflop computing systems is a significant programming challenge. One approach is to structure applications with an upper layer of many loosely coupled coarse-grained tasks, each comprising a tightly-coupled parallel function or program. “Many-task” programming models such as functional parallel dataflow may be used at the upper layer to generate massive numbers of tasks, each of which generates significant tightly coupled parallelism at the lower level through multithreading, message passing, and/or partitioned global address spaces. At large scales, however, the management of task distribution, data dependencies, and intertask data movement is a significant performance challenge. In this work, we describe *Turbine*, a new highly scalable and distributed many-task dataflow engine. Turbine executes a generalized many-task intermediate representation with automated self-distribution and is scalable to multi-petaflop infrastructures. We present here the architecture of Turbine and its performance on highly concurrent systems.

[*] Address for correspondence: Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL USA

Keywords: dataflow language, Swift, ADLB, MPI, Turbine

1. Introduction

Developing programming solutions to help applications utilize the high concurrency of multi-petaflop computing systems is a challenge. Languages such as Dryad, Swift, and Skywriting provide a promising direction. Their implicitly parallel dataflow semantics allow the high-level logic of large-scale applications to be expressed in a manageable way while exposing massive parallelism through many-task programming. Current implementations of these languages, however, limit the evaluation of the dataflow program to a single-node computer, with resultant tasks distributed to other nodes for execution.

We propose here a model for *distributed-memory* evaluation of dataflow programs that spreads the overhead of program evaluation and task generation throughout an extreme-scale computing system. This execution model enables function and expression evaluation to take place on any node of the system. It breaks parallel loops and concurrent function invocations into fragments for distributed execution. The primary novel features of our workflow engine — use of distributed memory and message passing — enable the scalability and task generation rates needed to efficiently utilize future systems.

We also describe our implementation of this model: Turbine. This paper demonstrates that Turbine can execute Swift programs on large-scale, high performance computing (HPC) systems.

1.1. Context

Exaflop computers, capable of 10^{18} floating-point operations/s, are expected to provide concurrency at the scale of $\sim 10^9$ on $\sim 10^6$ nodes [4]; each node will contain extremely high levels of available task and data concurrency [32]. Finding an appropriate programming model for such systems is a significant challenge, and successful solutions may not come from the single-language tools currently used but instead may involve hierarchical programming models [34]. In the many-task model, a logical application is composed from a time-varying number of interacting tasks [30]. Methodologies such as rational design, uncertainty quantification, parameter estimation, and inverse modeling all have this many-task property. All will frequently have aggregate computing needs that require exascale computers [33].

Running many-task applications efficiently, reliably, and easily on large parallel computers is challenging. The many-task model may be split into two important processes: *task generation*, which evaluates a user program, often a dataflow script, and *task distribution*, which distributes the resulting tasks to workers. The user work is performed by leaf functions, which may be implemented in native code or as external applications. Leaf functions themselves may be multicore or even multinode tasks. This computing model draws on recent trends that emphasize the identification of coarse-grained parallelism as a first distinct step in application development [24, 37, 39]. Additionally, applications built as workflows of many tasks are highly adaptable to complex, fault-prone environments [11].

1.2. Current approaches

Currently, many-task applications are programmed in two ways. In the first, the logic associated with the different tasks is integrated into a single program, and the tasks communicate through MPI messaging. This approach uses familiar technologies but can be inefficient unless much effort is spent incorporating efficient load-balancing algorithms into the application. Additionally, multiple other challenges arise, such as how to access memory from distributed compute nodes, how to efficiently access the file system, and how to develop a logical program. The approach can involve considerable programming effort if multiple existing component codes have to be tightly integrated.

Load-balancing libraries based on MPI, such as the Asynchronous Dynamic Load Balancing Library (ADLB) [23], or on Global Arrays, such as Shared Collections of Task Objects (Scioto) [12], have recently emerged as promising solutions to aid in this approach. They provide a master/worker system with a put/get API for task descriptions, thus allowing workers to add work dynamically to the system. However, they are not high productivity systems, since they maintain a low-level programming model and do not provide the high-level logical view or data model expected in a hierarchical solution.

In the second approach, a script or workflow is written that invokes the tasks, in sequence or in parallel, with each task reading and writing files from a shared file system. Examples include Dryad [17], Skywriting [25], and Swift [40]. This is a useful approach especially when the tasks to be coordinated are existing external programs. Performance can be poor, however, since existing many-task scripting languages are implemented with *centralized evaluators*.

1.3. Turbine: A scalable dataflow engine

```
1   Model m[];
2   Analysis a[];
3   Validity v[];
4   Plot p[];
5   int n;
6   foreach i in [0:n] {
7     // run model with random seed
8     m[i] = runModel(i);
9     a[i] = analyze(m[i]);
10    v[i] = validate(m[i]);
11    p[i] = plot(a[i], v[i]);
12  }
```

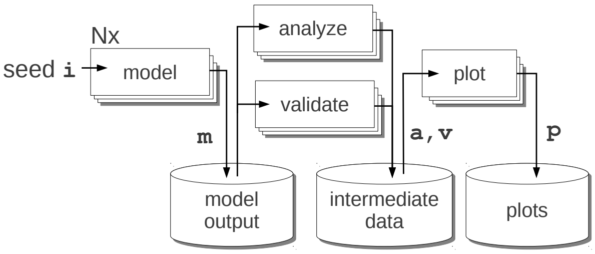


Figure 1. Swift example and corresponding dataflow diagram.

Consider an example application in Swift shown in Figure 1. The parallel `foreach` loop on line 6 runs N independent instances of the model. In a dataflow language such as Swift, this loop generates N concurrent loop iterations, each of which generates four user tasks with data dependencies. In previous implementations, the *evaluation* of the loop itself takes place on a single compute node (typically a cluster "login node"). Such nodes, even with many cores (today ranging from 8 to 24) are able to *generate* only about 500 tasks/s. Recently developed distribution systems such as Falkon [30] can *distribute* 3,000 tasks/s if the tasks are generated and enumerated in advance.

Despite the fact that the Swift language exposes abundant task parallelism, until now *the evaluation of the language-level constructs has been constrained to take place on a single compute node*. Hence, task generation rates can be a significant scalability bottleneck. Consider a user task that occupies a whole node for 10 seconds. For an application to run 10^9 such tasks across 10^6 nodes, 10^5 tasks must be initiated and distributed per second to keep that many cores fully utilized. *This is orders of magnitude greater than the rates achievable with single-node, centralized dataflow language evaluation.*

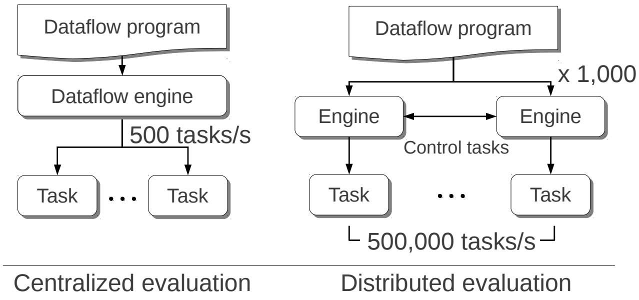


Figure 2. Left: prior centralized evaluator; Right: distributed evaluation with Turbine.

We describe here a model capable of generating and distributing tasks at this scale. Our implementation, the distributed *Turbine* engine, allocates a small fraction of the system as control processes that cooperate to rapidly evaluate the user script. Its innovation is based on expressing the semantics of parallel dataflow in a language-independent representation, with primitives amenable to the needs of the Swift parallel scripting language, and implementing a distributed evaluation engine for that representation that decentralizes the overhead of task generation, as illustrated in Figure 2. Turbine execution employs distributed dependency processing, task distribution by a previously developed load balancer, and a distributed in-memory data store that makes script variables accessible from any node of a distributed-memory system. Turbine thus combines the performance benefits of explicitly parallel asynchronous load balancing with the programming productivity benefits of implicitly parallel dataflow scripting.

1.4. Paper organization

The remainder of this paper is organized as follows. In §2, we motivate this work by providing two representative examples of scripted applications. In §3, we describe the programming and task distribution models on which our work is based. In §4, we describe our design for parallel evaluation of Turbine programs. In §5, we present the implementation of the Turbine engine framework. In §6, we report performance results from the use of the implementation in various modes. In §7, we discuss other related work. In §8, we offer concluding remarks.

2. Motivation: Applications

This section presents many-task applications that may be conveniently represented by the parallel scripting paradigm. We motivate our work by demonstrating the need for a highly scalable many-task programming model.

2.1. Power-grid distribution

Solving problems in power grid economic dispatch is of critical importance when evaluating the feasibility of future-generation energy production and distribution. In one approach, the problem is represented as a massive integer programming challenge, where the integer-valued results represent whether a particular resource is used. The solution is computed to optimize the probable cost over a large range of potential scenarios defined by weather, demand, and so on. An example power grid for the state of Illinois is shown in Figure 3. Vertices in the power grid graph represent power consumers and producers; producers may be toggled on or off to satisfy demand under varying consumer scenarios.

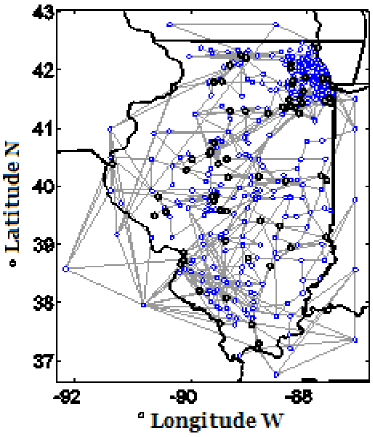


Figure 3. Diagram of power grid (by Victor Zavala).

PIPS [22, 21, 28] is a massively parallel program that solves this problem by using stochastic programming. In the present work, we extend the PIPS project by evaluating the cost of the solution produced by PIPS in a large number of scenarios. The latter is a naturally parallel problem easily addressed by the Turbine model and the Swift language. Our PIPS application is presented further in Section 6.2.

2.2. Highly parallel land use modeling

The focus of the Decision Support System for Agrotechnology Transfer (DSSAT) application is to analyze the effects of climate change on agricultural production [18]. Projections of crop yields at regional scales are carried out by running simulation ensemble studies on available datasets of land cover, soil, weather, management, and climate. The computational framework starts with the DSSAT crop systems model and evaluates this model in parallel by using Swift. Benchmarks have been performed on prototype simulation *campaigns*, measuring yield and climate impact for a single crop (maize) across the conterminous United States (120k cells) with daily weather data and climate model output spanning 120 years (1981-2100) and 16 different configurations of fertilizer, irrigation, and cultivar choice. Figure 4 shows a listing of the DSSAT application in Swift. It represents the prototype simulation carried out for four different campaigns, each running 120,000 jobs. In the future, DSSAT is expected to run at a global scale, cover more crops, and consume 3-4 orders of magnitude more computation time.

```
app (file output) RunDSSAT (file input[]) {
  RunDSSAT @output @input ;
}

string campaigns[] = ["srb","ncep","cpc","cpc_srb"];
string glst[] = readData("gridList.txt");

foreach c in campaigns {
  foreach g, i in gridList {
    file out <strcat(cmp, "/", glst[i], ".tar")>;
    file in[] <strcat(c, "/", glst[i], "/*.*")>;
    out = RunDSSAT(in);
  }
}
```

Figure 4. Swift code for DSSAT application.

2.3. Scientific collaboration graph analysis

The SciColSim project assesses and predicts scientific progress through a stochastic analysis of scientific collaboration networks [14]. The software mines scientific publications for author lists and other indicators of collaboration. The strategy is to devise parameters whose values model real-world human interactions. These parameters are governed by an "evolve" function that performs a simulated annealing. A "loss" factor is computed on each iteration of annealing denoting the amount of effort expended.

```
foreach i in innovation_values {          // O(20)
  foreach r in repeats {                  // 15
    iterate cycle in annealing_cycles {   // O(100)
      iterate p in params {               // 3
        foreach n in reruns {             // 1000
          evolve(...); // 0.1 to 50 seconds
        }}}}}
```

Figure 5. Swift code for SciColSim application.

Swift-like pseudocode for this application is shown in Figure 5. Note that Swift's `foreach` statement indicates a parallel loop, whereas the `iterate` statement indicates a sequential loop. The computation involves a repetition of annealing cycles over a range of innovation values recomputed for increased precision. The number of jobs as a result of these computations is on the order of 10 million for a single production run, with complex dependencies due to the interaction of the loop types.

2.4. Other applications

In addition to the mentioned applications, ensemble studies involving different methodologies such as uncertainty quantification, parameter estimation, massive graph pruning and inverse modeling all require the ability to generate and dispatch tasks on 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 fragments by using Matlab scripts on hundreds of cores. This application will utilize tens of thousands of cores or more in the future. SWAT is a motivator for our work because of the large number of small data objects, currently stored in files. *Biomolecular analysis* via ModFTDock results in a large quantity of available tasks [16] and represents a complex, multistage workflow.

Table 1. Quantitative description of applications and projection of required performance on 10^6 cores.

Application	Stage	Measured		Required	
		Tasks	Task Duration	Tasks	Task Rate
Power-grid Distribution	PIPS rounding	100,000	60 s	10^9	1.6×10^4/s
DSSAT	runDSSAT	500,000	12 s	10^9	8.3×10^4/s
SciColSim	evolve	10,800,000	10 s	10^9	10^5/s
SWAT	swat	2,200	3-6 h	10^5	46-93/s
modftdock	dock	1,200,000	1,000 s	10^9	10^3/s
	modmerge	12,000	5 s	10^7	2×10^5/s
	score	12,000	6,000 s	10^7	166/s

Summary: Table 1 shows a summary of 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. Turbine was designed based on the computational properties these applications display, and we intend that performance and resource utilization will increase when these applications are run on the new system.

3. Programming and task Distribution Models: Swift and ADLB

The work described here builds on the Swift parallel scripting programming model, which has been used to express a wide range of many-task applications, and ADLB, which provides the underlying task distribution framework for Turbine.

Swift [40] is a parallel scripting language for scientific computing. The features of this typed and concurrent language that support distributed scientific batch computing include data structures (arrays, structures), string processing, use of external programs, and external data access to filesystem structures. The current Swift implementation compiles programs into the Karajan workflow language, which is interpreted by a runtime system based on the Java CoG Kit [36]. While Swift can generate and schedule thousands of tasks and manage their execution on a wide range of distributed resources, each Swift script is evaluated on one node, resulting in a performance bottleneck. Removing such bottlenecks is the primary motivation for the work described here.

ADLB [23] is an MPI-based library for managing a large, distributed work pool. ADLB applications use a simple put/get interface to deposit "work packages" into a distributed work pool and retrieve them. Work package types, priorities, and "targets" allow this interface to implement sophisticated variations of the classical master/worker parallel programming model. ADLB is efficient (can deposit up to 25,000 work packets per second per node on an Ethernet-connected Linux cluster) and scalable (has run on 131,000 cores on an IBM BG/P for nuclear physics applications [23]). ADLB is used as the load-balancing component of Turbine, where its work packages are the indecomposable tasks generated by the system. The implementation of its API is invisible to the application (Turbine, in this case), and the work packages it manages are opaque to ADLB. An experimental addition to ADLB has been developed to support the Turbine data store, implementing a publish/subscribe interface. This allows ADLB to support key features required for Swift-like dataflow processing.

4. Scalable Distributed Dataflow Processing

We describe here the evaluation model that Turbine uses to execute implicitly concurrent dataflow programs. Turbine's function is to interpret an intermediate representation of a dataflow program on a distributed-memory computing resource. The main aspects of the Turbine model are as follows.

Implicit, pervasive parallelism. Most statements are relations between single assignment variables. Dynamic data-dependency management enables expressions to be evaluated and statements executed when their data dependencies are met. Thus, all statements are eligible to be run concurrently as dataflow allows.

Typed variables and objects. Variables and objects are constructed with a simple type model comprising the typical primitive scalar types, a container type for implementing arrays and structures, and a file type for external files and external in-memory variables.

Constructs to support external execution. Most application-level work is performed by external user application components (programs or functions); Turbine is primarily a means to carry out the composition of the application components.

4.1. The Swift programming model

To underscore the motivation for this work, we briefly summarize the Swift program evaluation model [40]. A Swift program starts in a global scope in which variables may be defined. Attempting to assign to a scalar variable more than once is an error detectable at compile time. All statements in the scope are allowed to make progress concurrently, in dataflow order. Statements that must wait on input variables are recorded and entered into a data structure that is notified when the inputs are stored. Upon notification, the statement is started. When variables are returned by functions, they are closed. Input and output variables passed to functions exist in the original caller's scope and are passed and accessed by reference. Since input variables cannot be modified in the called function, they behave as if passed by value.

A key feature of Swift is the distinction between composite functions and leaf functions (denoted with the app keyword). Composite invocations cause the creation of new scopes (stack frames) in which new local variables are dynamically created and new statements may be issued. Leaf functions are used to launch external execution. Originally, leaf functions were used primarily to launch remote execution by using grid computing techniques. The new model, using Turbine, is designed primarily to link with and call user functions directly.

The original Swift system can support scripts that run on thousands of cores. However, the evaluation of the script itself is constrained to run on a single node. This limitation is in large part due to the fact that Swift uses data structures in a shared address space for data dependency tracking; no mechanism exists for cross-address-space communication and synchronization of the state of future objects. The execution model of Turbine eliminates the limitations of this centralized evaluation model.

4.2. Overview of Turbine features

The high-level architecture of Turbine is shown in Figure 6. The center of the system is the network of ADLB ***servers***, which provide load balancing and data services. ***Engines*** evaluate the user Turbine code and its data dependencies, generating and submitting tasks to other engines and to ***workers*** that execute external code such as user programs and functions. Tasks may store data, resulting in notifications to listening engines, allowing them to satisfy data dependencies and release more tasks. The Swift program compiler is presented elsewhere [2, 41].

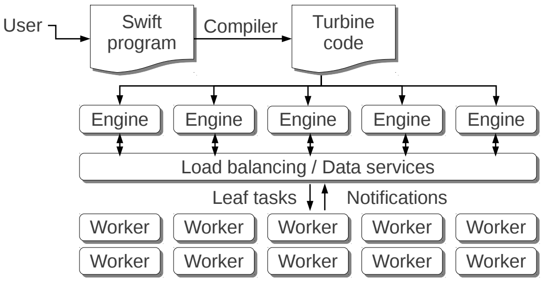


Figure 6. Turbine component architecture. (ADLB server processes are opaque to Turbine.)

Turbine engines implement the key dataflow operations necessary to carry out data-dependent processing. Scalability is achieved by implementing this work in a location-independent manner, and distributing this work among the engines through ADLB. The engine processes operate as ordinary ADLB clients, communicating with other engines and workers through tasks.

Turbine operations include data manipulation, the definition of data-dependent execution expressed as data-dependent *statements*, and higher-level constructs such as loops and function calls. Programs consist of *composite functions*, which are essentially a context in which to define a body of statements. *Leaf functions* are provided by the system as built-ins or may be defined by the user to execute as external (leaf) tasks; this is the primary method of performing user work. Generally, composite functions are executed by engines, and leaf functions are executed by workers. Each statement calls a composite or leaf function on input and output variables. Thus, statements serve as links from task outputs to inputs, forming an implicit, distributed task graph as the script executes.

Variables are typed in-memory representations of user data. Variable values may be scalars, such as integers or strings or containers representing language-level features, such as arrays or structures (records). All variables are single-assignment futures: they are open when defined and closed after a value has been assigned. Containers are also futures, as are each of their members. Containers can be open or closed. While they are open, their members can be assigned values. A container is closed by

Turbine when all program branches eligible to modify the container complete. In practice, this action is controlled in the translation from the source (Swift) program by scope analysis and the detection of the last write, the end of the scope in which it is defined, since any returned containers must be closed output variables. *Scopes* (i.e., stack frames) provide a context for variable name references. Scopes are composed into a linked call stack. Execution continues until all statements complete.

Script variables are stored in a globally accessible data store provided by the servers. Turbine operations are available to store and retrieve data to local memory as in a typical load-store architecture.

Data dependencies for statements are stored locally on the engine that issued the statement. Engines receive notifications from the global data store in order to make progress on data-dependent statements when data is stored, regardless of which process stored the data.

Thus, the main contribution of Turbine that facilitates distributed evaluation is a *distributed variable store based on futures*. This store, accessible from any process within a Turbine program, enables values produced by a function executing on one node to be passed to and consumed by a function executing on another; the store manages the requisite event subscriptions and notifications. Examples of the use of the distributed future store follow in the next section.

4.3. Turbine use cases

In this section, we enumerate critical high-level dataflow language features as expressed by Swift examples. We demonstrate that our highly distributed evaluation model is capable of implementing a general-purpose dataflow language like Swift.

Basic dataflow. Turbine statements that require variables to be set before they can execute (primarily the primitives that call an external user application function or program) are expressed as the target operations of statements that specify the action, the input dependencies required for the action to execute, and the output data objects that are then set by the action.

Consider the fragment derived from the example in Figure 1:

```
Model m; // ... etc.
m = runModel();
a = analyze(m);
v = validate(m);
p = plot(a, v);
```

Example 1(a): Swift

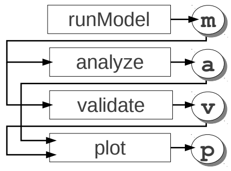
This fragment assumes variables a, v, and p are to be returned to a calling procedure. The input and output data dependencies are captured in the following four Turbine statements:

```
allocate m # ... etc.
call_app runModel [ m ] [ ]
call_app analyze  [ a ] [ m ]
call_app validate [ v ] [ m ]
call_app plot     [ p ] [ a v ]
```

Example 1(b): Turbine

These statements correspond to data dependencies stored in the process that evaluated them. Data definitions correspond to addresses in the global data store. The runModel task may result in a task issued to the worker processes; when `m` is set by the worker as a result of the first action, the engines responsible for dependent actions are notified, and progress is made.

Each Turbine engine maintains a list of data-dependent statements that have been previously read. Additionally, variable addresses are cached. Thus, a sequence of statements as given in Example 1(b) results in an internal data structure that links variable addresses to statement actions. As shown in the figure below, the `runModel` task has no dependencies. Statements `analyze` and `validate` refer to data `m`, which the engine links to the execution of these statements.



Expression evaluation. In practice, scripting languages provide much more than the coordination of external execution. Because Swift and other higher-level workflow languages offer convenient arithmetic and string operations, processing expressions is a critical feature. The following Swift fragment shows the declaration of two integers, their addition, and the printed result via the built-in `trace`.

```
int i = 3, j = 4, k;
k = i + j;
trace(k);
```

Example 2(a): Swift

At the Turbine level, this results in the declaration of three script variables and two statements. At run time, all five lines are read by the Turbine engine. Variable `k` is open. Statement `plus_integer`, a built-in, is ready because its inputs, two literals, are closed. The engine executes this function, which closes `k`. The engine notifies itself that `k` is closed, which makes statement `trace` ready; it is then executed.

```
allocate i integer 3
allocate j integer 4
allocate k integer
call_builtin plus_integer [ i j ] [ k ]
call_builtin trace [ k ] [ ]
```

Example 2(b): Turbine

Conditional execution. Turbine offers a complete set of features to enable high-level programming constructs. Conditional execution is available in Swift in a conventional way. The following example shows that the output of a user function may be checked for a condition, resulting in a context block for additional statements:

```
c = extractStatistic(a);
if (c) {
  trace("Warning: c is non-zero");
}
```

Example 3(a): Swift

At the Turbine level, the conditional statement is dependent on a single input, the result of the conditional expression. It essentially executes as a built-in that conditionally jumps into a separate context block. This block is generated by the compiler from the body of the conditional construct. Code executing in the body block references variables as though it were in the original context.

```
    ...   # open code
    call_app extractStatistic [ a ] [ c ]
    statement [ c ] if-1 [ c ]
}

proc if-1 { c } {
    set v:c [ get_integer c ]
    if (v:c) {
        allocate s string "Warning: c is non-zero"
        call_builtin trace [ ] [ s ]
    }
}
```

Example 3(b): Turbine

Our compiler is capable of translating `switch` and `else if` expressions using this basic technique.

Since the variables are accessible in the global data store, and since the condition could be blocked for a long time (e.g., if `extractStatistic` is expensive), the condition body block could be started on another engine. Since the dependencies here are essentially linear, however, shipping the execution would yield no scalability benefit and would generate additional traffic to the load-balancing layer. Consequently, our model does not distribute conditional execution.

Composite functions. As discussed previously, there are two main types of user functions: composite functions and leaf functions. Leaf functions are opaque to the dataflow engine and are considered in §5. Composite functions provide a context in which to declare data and issue statements. Since statements are eligible to be executed concurrently, composite function call stacks enable concurrency.

Example 4(a) shows part of the recursive implementation of the nth Fibonacci number in Swift syntax. The two recursive calls to `fib` may be executed concurrently. Thus, Turbine packs these statements as work units tagged for execution on peer engines as made available by the load balancer.

```
(int f) fib(int n) {
  if (n > 2)
    f = fib(n-1) + fib(n-2);
  ...
}
```

Example 4(a): Swift

The translated Turbine code below demonstrates that after the arithmetic operations are laid out, `call_composite` is used to issue the two recursive calls on *different* engines as selected by the load balancer. When such a work unit is received by another engine, it unpacks the statement and its attached data addresses (e.g., `f` and `n`), jumps into the appropriate block, and evaluates the statements encountered there.

```
proc fib { n f } {
    allocate t0 integer 1
    allocate t1 integer
    allocate t2 integer
    call_builtin minus_integer [ t1 ] [ n t0 ]
    # fib(n-1)
    call_composite fib [ t2 ] [ t1 ]
    ...
    # fib(n-2)
    call_composite fib ...
    call_builtin plus_integer [ f ] [ t2 ... ]
    ...
}
```

Example 4(b): Turbine

Data structures. Swift contains multiple features for structured data, including C-like arrays and structs. Swift arrays may be treated as associative arrays; the subscripts may be strings, and so forth. Structured data members are linked into the containing data item, which acts as a table. These structures are fully recursive. Structured data variables may be treated as futures, enabling statements that are dependent on the whole variable. Swift arrays and structs are allocated automatically as necessary by the compiler and are closed automatically by the runtime system when no more modifications to the data structure are possible. Swift limits the possible modifications to a structured data item to the scope in which it was declared; in other scopes, the data is read-only.

As shown, `eye2` builds a 2 × 2 array. The data structure `a` is allocated by the compiler since it is the function output.

```
(int a[][])  eye2() {
  a[0][0] = 1;
  a[0][1] = 0;
  a[1][0] = 0;
  a[1][1] = 1;
}
```

Example 5(a): Swift

In the Turbine implementation, a reusable *containers* abstraction is used to represent linked data structures. A container variable, stored in a globally accessible location, allows a user to insert and look up container subscripts to store and obtain data addresses for the linked items.

In the example below, multiple containers are allocated by the `allocate_container` statement, including the top-level container `a` and the container `t2=a[0]`, etc. The `container_insert` command inserts `t1` at `a[0]`, and so on.

```
proc eye2 { a } {
    allocate_container a
    allocate_container t1
    allocate_container t2
    allocate i0 integer 0
    allocate i1 integer 1
    container_insert_imm a 0 t1
    container_insert_imm t1 0 i0
    container_insert_imm t1 1 i1
    ...
}
```

Example 5(b): Turbine

When eye2 returns, a is closed, and no further changes may be made. Thus, a user statement could be issued that is dependent on the whole array.

Iterations. The primary iteration method in Swift is the foreach statement, which iterates over an array. At each iteration, the index and value are available to the executing block. In the example below, the user transforms each a[i] to b[i] via f:

```
int b[];
foreach i, v in a {
  b[i] = f(a[i]);
}
```

Example 6(a): Swift

Each execution is available to be run concurrently, that is, when each a[i] is closed. Thus, we reuse our block statement distribution technique on a compiler-generated block:

```
  ... # open code
  allocate_container b
  loop a [ a ] loop_1
}

# inputs: loop counter, loop variable and additionals
proc loop_1 { i v a } {
    # t1 is a local value
    set t1 [ container_lookup_imm [ a i ] ]
    allocate t2 integer
    call_composite f [ t2 ] [ t1 ]
    container_insert_imm b i t2
}
```

Example 6(b): Turbine

The built-in loop statement retrieves the set of available indices in a; for each index, a work unit for block loop_1 is sent to the load balancer for execution on a separate engine. Thus, data dependencies created by statements in each iteration are balanced among those engines.

An example of this process is shown in Figure 7. First, the user script is translated into the Turbine format ①. Following Example 6(b), a loop statement is evaluated, resulting in the interpretation of a distributed loop operation, producing additional new work units containing script fragments ②. These fragments are distributed by using the load balancer system ③ and are evaluated by engines elsewhere, resulting in the evaluation of application programs or functions, which are blocks of Turbine statements ④. These blocks execute, producing data-dependent expressions for their respective local engines.

5. Turbine Engine Implementation

In this section, we describe our prototype implementation of Turbine, which implements the model described in the previous section.

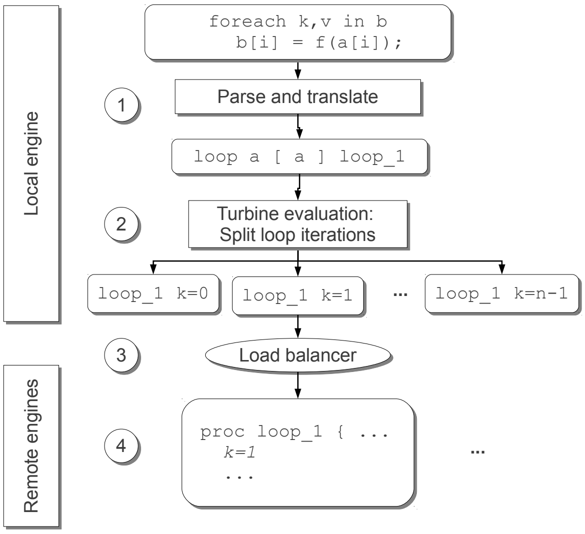


Figure 7. Distributed iteration in Turbine.

5.1. Program structure

In order to represent the functionality described in §4, the Turbine implementation consists of the following components:

- A compiler to translate the high-level language (e.g., Swift) into its Turbine representation
- The load-balancing and data access services
- The data-dependency engine logic
- Built-ins and features to perform external execution and data operations

The source program (e.g., in Swift) is translated into its Turbine representation by a provided compiler, described elsewhere [2].

In our prototyping work, we leverage Tcl [38] as the implementation language and express available operations as Tcl extensions. We provide a Tcl interface to ADLB to operate as the load balancer. The load balancer was extended with data operations; thus, in addition to task operations, data storage, retrieval, and notification operations are available. ADLB programs are standard MPI programs that can be launched by `mpiexec` (see §3). Generated Turbine programs are essentially ADLB programs in which some workers act as data-dependency processing engines. Core Turbine features were implemented as a C-based Tcl extension as well. Tcl features are used simply to represent the user script and not to carry out performance-critical logic.

The engines cooperate to evaluate the user script using the techniques in §4 to produce execution units for distribution by the distribution system. The engines use the ADLB API to create new tasks and to distribute the computational work involved in evaluating parallel loops and composite function calls — the two main concurrent language constructs.

ADLB performs highly scalable task distribution but does incur some client overhead and latency to ingest tasks. As more ADLB client processes are employed to feed ADLB servers, this overhead becomes more distributed, and the overall task ingestion and execution capacity increases. Each client is attached to one server, and traffic on one server does not congest other server processes. ADLB can scale task ingestion fairly linearly with the number of servers on systems with hundreds of thousands of cores. Thus, a primary design goal of the Turbine system is to distribute the work of script evaluation to maximize the ADLB task ingestion rate and hence the utilization of extreme-scale computing systems.

The user defines leaf functions to perform compute-intensive work on workers. Tools are provided to call native C/C++ functions from Swift and Turbine, implemented through SWIG [5] and shared library techniques. To call an external program, the user simply calls Tcl's `exec` as a leaf function. Users may add data-dependent functions to access data, etc.

5.2. Distributed data storage in Turbine

The fundamental system mechanics required to perform the operations required by the previous section were developed in Turbine, a novel *distributed future store*. The Turbine implementation comprises a globally addressable data store, an evaluation engine, a subscription mechanism, and an external application function evaluator. Turbine script variable data is stored on servers and processed by engines and workers. These variables may be `string`, `integer`, `float`, `file`, `blob`, or `container` data.

Variable addresses in Turbine are represented as 64-bit integers. Addresses are mapped to responsible server ranks through a simple hashing scheme. Unique addresses may be obtained from servers. Given an address, a variable may be allocated and initialized at that location. An initialized variable may be the target of a notification request by any process. A variable may be set once with a value, after which the value may be obtained by any process.

Progress is made when futures are set and engines receive notification that new input data is available. Turbine uses a simple subscription mechanism whereby engines notify servers that they must be notified when a data item is ready. As a result, the engine either 1) finds that the data item is already closed or 2) is guaranteed to be notified when it is. When a data item is closed, the closing process receives a list of engines that must be notified regarding the closure of that item, which allows dependent statements to progress.

A variable of type `file` is associated with a string file name; however, unlike a `string`, it may be read before the file is closed. Thus, output file locations may be used in Turbine statements as *output* data, but the file name string may be obtained in order to enable the creation of a shell command line.

```
allocate a file "input.txt"
allocate b file "output.txt"
call_app create_file [ a ] [   ]
call_app copy_file   [ b ] [ a ]
```

At the end of this code fragment, `output.txt` is closed, allowing it to be used as the input to other Turbine tasks. Note that this file-type variable does not represent the contents of the file; it is simply the future variable corresponding to the existence of the file with the given name. The user leaf task (here, `copy_file`) is responsible for carrying out I/O.

Blob (binary large object) variables are arbitrary binary datasets implemented as C byte arrays. They are represented in Turbine by a pointer+length record and are globally addressable. These are used primarily for passing data to and from user C/C++ leaf functions.

Container variables are similar to other Turbine script variables, but the value of a container variable is a mapping from keys to Turbine variable addresses. Operations are available to insert, lookup, and enumerate container contents.

6. Performance Results

In this section, we provide synthetic performance results from Turbine benchmarks and application results from the PIPS application.

6.1. Turbine benchmarks

Our benchmark results focus on three main aspects of Turbine performance: task distribution using ADLB, data operations using the new ADLB data services, and evaluation of the distributed Turbine loop construct. These demonstrate the ability of Turbine to meet the performance goals required by our applications.

In each case, we present results from systems of Turbine control processes. We neglect worker processes because the focus of this section is our distributed-memory dataflow evaluation functionality and not task distribution. All presented results were obtained on the SiCortex 5872 at Argonne. Each MPI process was assigned to a single core of a six-core SiCortex node, which runs at 633 MHz and contains 4 GB RAM. The SiCortex contains a proprietary interconnect with ∼1 microsecond latency.

6.1.1. Raw task distribution

To evaluate the ability of the Turbine architecture to meet its performance requirements, we first report the performance of ADLB. Following that model, each ADLB server occupies one control process. Thus, we desire to measure the task throughput rate of a single ADLB server. We configured ADLB with one server and a given number of workers. A single worker reads an input file containing a list of tasks for distribution over ADLB to other workers. This emulates a Turbine use case with a single engine that can produce tasks as fast as lines can be read from an input file. Two cases are measured: one in which workers execute `sleep` for a zero-duration run (labeled "/bin/sleep") and one in which they do nothing (labeled "no-op").

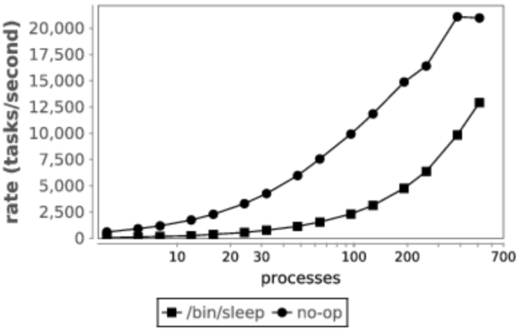


Figure 8. Task rate result for ADLB on SiCortex.

The results are shown in Figure 8. In the no-op case, for increasing numbers of client processes, performance improved until the number of clients was 384. At that point, the ADLB server was processing 21,099 tasks/s, far exceeding the desired rate of 1,000 tasks/s per control process. The no-op performance is then limited by the single-node performance and does not increase as the number of clients is increased to 512. When the user task actually performs a potentially useful operation such as calling an external application (`/bin/sleep`), the single-node ADLB server performance is not reached by 512 client processes.

We note that in practice a Turbine application may post multiple "system" tasks through ADLB in addition to the "user" tasks required for the user application. Thus, the available extra processing on the ADLB server is appropriate. Overall, this test indicates that ADLB can support the system at the desired scale.

6.1.2. Data operations

Next, we measure another key underlying service used by the Turbine architecture: the ADLB-based data store. This new component is intended to scale with the number of tasks running in the system; each task will need to read and write multiple small variables in the data store in order to enable the user script to make progress.

We configured a Turbine system with a given number of servers and clients and with one engine that is idle. ADLB store operations are performed on each client, each working on independent data. Each client interacts with different servers on each operation. Each client creates 200,000 small data items in a two-step process compatible with the dataflow model; they are first allocated and initialized, then set with a value and closed.

Figure 9 shows that for increasing numbers of servers and clients, the insertion rate increases slightly sublinearly. In the largest case, 1,024 servers were targeted by 1,023 workers with 1 idle engine, achieving an insertion rate of 19,883,495 items/s. Since data operations are independent, congestion occurs only when a server is targeted by multiple simultaneous operations, creating a temporary hot spot. At a performance target of 1,000 tasks/s per server, this allows each task to perform almost 20 data operations without posing a performance problem.

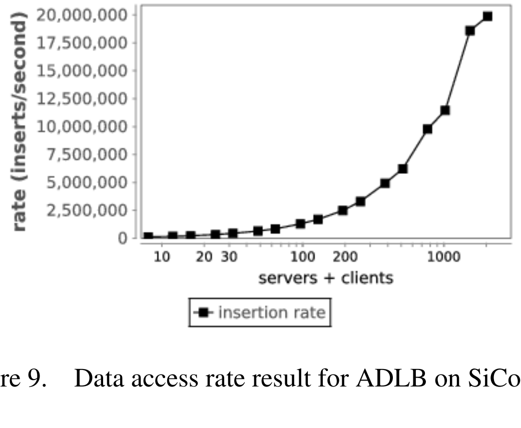


Figure 9. Data access rate result for ADLB on SiCortex.

The data item identifiers were selected randomly; thus the target servers were selected randomly. With the existing Turbine and ADLB APIs, an advanced application could be more selective about data locations, eliminating the impact of hot spots. This case does not measure the performance of data retrieval operations, covered implicitly in §6.1.4.

6.1.3. Distributed data structure creation

Thus far we have measured only the performance of underlying services. Here, we investigate the scalability of the distributed engine processes. The engines are capable of splitting certain large operations to distribute script processing work. An important use case is the construction of a distributed container, which is a key part of dataflow scripts operating on structured data. The underlying operations here are used to implement Swift's range operator, which is analogous to the colon syntax in Matlab; for example, `[0:10]` produces the list of integers from 0 to 10. This can be performed in Turbine by cooperating engines, resulting in a distributed data structure useful for further processing.

We measured the performance of the creation of distributed containers on multiple engines. For each case plotted, a Turbine system was configured to use the given number of engines and servers in a 1:1 ratio. A Turbine distributed range operation was issued, creating a large container of containers. This triggered the creation of script variables storing all integers from 0 to $N \times 100,000$, where N is the number of Turbine engine processes. The operation was split so that all engines were able to create and fill small containers that were then linked into the top-level container, as diagrammed in Figure 10. Workers were not involved in this operation.

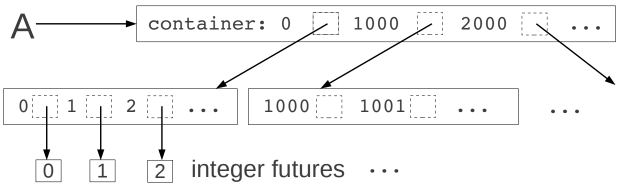


Figure 10. Distributed container data structure. Each container and integer may be stored on a different server.

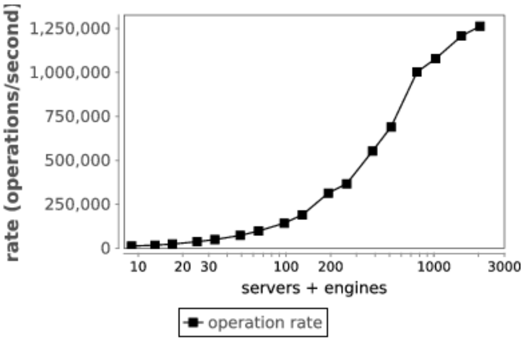


Figure 11. Range creation rate result for Turbine on SiCortex.

Figure 11 shows that the number of cooperating control processes, increasing to the maximal 2,048, half of which act as ADLB servers and half of which act as Turbine engines, does not reach a performance peak. Each operation creates an integer script variable for later use. At the maximum measured system size, the system created 1,262,639 usable script variables per second, in addition to performing data structure processing overhead, totaling 204,800,000 integer script variables in addition to container structures.

6.1.4. Distributed iteration

Once the application script has created a distributed data structure, it is necessary to iterate over the structure and use the contents as input for further processing. For example, once the distributed container is created in the previous test, it can be used as the target of a `foreach` iteration by multiple cooperating engines.

To measure the performance of the evaluation of distributed loops on multiple engines, for each case plotted, we configured a Turbine system to use the given number of engines and servers in a 1:1 ratio. In each case, a Turbine distributed range operation is issued, which creates a large container of containers. The operation is split so that all engines are able to create and fill small containers that are then linked into the top-level container. Then, the engines execute no-op tasks that read each entry in the container once. The measurement was made over the whole run, capturing range creation and iteration. Thus, each "operation" measured by the test is a complex process that represents the lifetime of a user script variable in a Turbine distributed data structure.

As in the previous test, each engine creates 100,000 variables and links them into the distributed container, over which a distributed iteration loop is carried out. Figure 12 shows that performance increases monotonically up to the largest measured system, containing 1,024 servers and 1,024 engines, in addition to 1 idle worker. At this scale, Turbine processes 566,685 operations per second.

Performance could be improved in multiple ways. First, this test was performed on the individually slow SiCortex processors. Second, we plan multiple optimizations to improve this performance. As noted in the previous test, variables are created in a multiple-step manner corresponding to a straightforward use of our dataflow model. Since we are creating "literal" integers, these steps could be replaced with composite operations that allocate, initialize, and set values in one step. Multiple literals could be simultaneously set if batch operations were added to the API. The loop-splitting algorithm itself is targeted for optimization and better load balancing. Third, an application could use proportionally more processes as control processes and fewer workers.

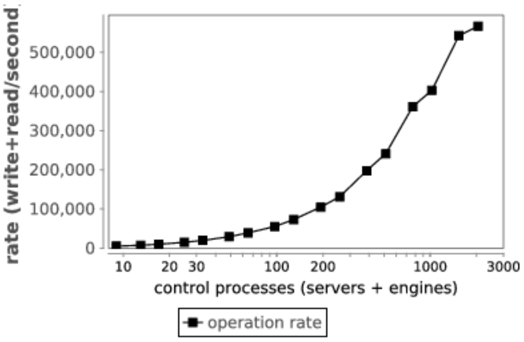


Figure 12. Range creation, iteration rate result for Turbine on SiCortex.

6.2. Application study: PIPS

An overview of the PIPS application was presented in Section 2.1.

A diagram of the power grid problem addressed here is in Figure 13. The data flow starts with the massively parallel PIPS numerics code, which produces a potential solution to the power grid problem (prior work). Then, the solution is applied to each scenario, represented by a single file. This produces massive task parallelism. Once all tasks have completed, a simple analysis is performed and aggregate results are returned to the user.

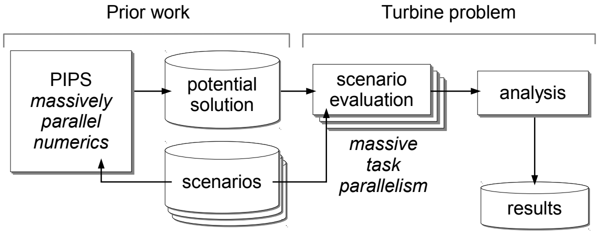


Figure 13. Diagram of power grid problem addressed by Turbine.

The corresponding Swift code is shown in Figure 14. Function names shown in bold indicate calls into the PIPS C++ API. In this short program, the solution is read in by using the PIPS call `readConvSolution()`, and stored in a blob. This blob is then processed by the PIPS `roundSolution()` function. Next, the scenarios are evaluated in parallel by the Swift parallel `foreach` loop. A simple sum of the cost results is computed when all entries into array `v` are complete. While this program does not flex complex Turbine data dependencies, it does test Turbine's ability to produce and manage a large amount of parallel work and correctly handle the array semantics for array `v`.

```
main () {
  string data = "/path/to/data";
  int nScenarios = 4096;
  blob s = readConvSolution(dataPath,solutionPath);
  float cutoff = 0.8;
  blob r = roundSolution(s,cutoff);
  float v[];
  foreach i in [0 : nScenarios-1] {
    v[i] = evaluateRecourseLP(data,nScenarios,i,r);
  }
  float result = sum(v);
  printf("result: %f\n", result);
}
```

Figure 14. Swift source code for power grid analysis (by Miles Lubin).

The PIPS code was linked with Turbine on the IBM Blue Gene/P *Challenger* at Argonne. Each MPI process was assigned to a single core of a four-core node, which runs at 850 MHz and contains 2 GB RAM. The Blue Gene/P contains a proprietary interconnect with ∼6 microsecond latency.

The running times of the calls to the computational function, `evaluateRecourseLP()`, are shown as a cumulative distribution function in Figure 15. As shown, most running times are between 70 and 80 seconds, however, some are well outside this range, making static scheduling undesirable. Thus, it is an interesting case for analyzing task generation and distribution in the Turbine and ADLB systems.

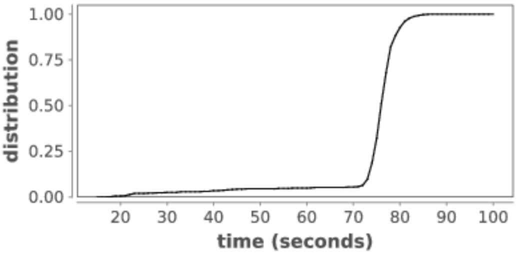


Figure 15. Distribution of task run times in desired PIPS run.

The corresponding Turbine code, as produced by the STC compiler, is shown in Figure 16. `nScenarios=4` is recognized as a global constant value (l. 1-5). The for loop (l. 13-15) results in rapid calls to the loop body, which is represented as another function (l. 20). Each pass through the loop body

```
1   swift:constants {
2       global i_4096
3       allocate i_4096 integer
4       store_integer i_4096 4096
5   }
6   swift:main {
7       allocate data string ...
8       allocate s blob
9       allocate r blob
10      allocate_container v integer
11      allocate result float
12      ... #  initial functions omitted
13      for { i=0 } { i <= i_4096 } { increment i } {
14          slot_create v
15          rule loop0:body [ ] [ v data r i_4096 i ]
16      }
17      slot_drop v
18      rule sum [ result ] [ v ]
19  }
20  loop0:body { data v r i_4096 i } {
21      allocate t float
22      rule swiftpips::evaluateRecourseLP [ t ] [ data r i_4096 i ]
23      container_insert v i t
24      slot_drop v
25  }
```

Figure 16. Generated Turbine code for power grid analysis.

results in the creation of a data-dependent execution of `evaluateRecourseLP()` (l. 22); its result is stored in array `v`. When all slots on array `v` have been dropped, the `sum` statement is triggered, producing the final result.

6.2.1. PIPS performance

The performance of this program is shown in Figure 17. Our initial performance metric is *scaling*: the ability to increase the computation rate as the number of processor cores increases. The computation rate is defined in terms of the number of calls to the compute-intensive user function (`evaluateRecourseLP()`) per unit time. Starting with 32 cores, we successively doubled the number of cores up to 1,024. One engine was used in each case. The ideal scaling would double the call rate at each step. Any difference between the ideal call rate and the measured rate is due to Turbine overhead. As shown, there is no measured drop in call rate due to Turbine overhead at this scale.

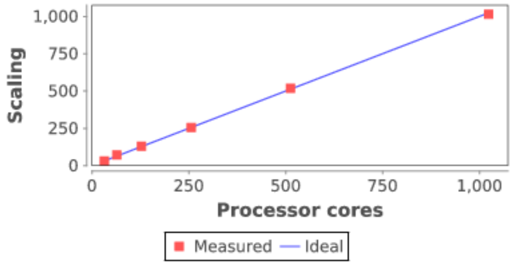


Figure 17. User function call rate scaling for PIPS.

To investigate PIPS/Turbine performance further, we applied MPE logging [8] and Jumpshot to profile Turbine behavior during the run. In Figure 18, we show the Jumpshot diagram for a 32-process run. Task transition times (Turbine overhead) between tasks are not visible at this zoom level, indicating very good utilization. As shown, however, the end of the workflow is not abrupt, and the final tasks finish in a ragged pattern. This causes a loss in utilization due to trailing tasks [3]. An approach to this problem with STC features is presented elsewhere [41].

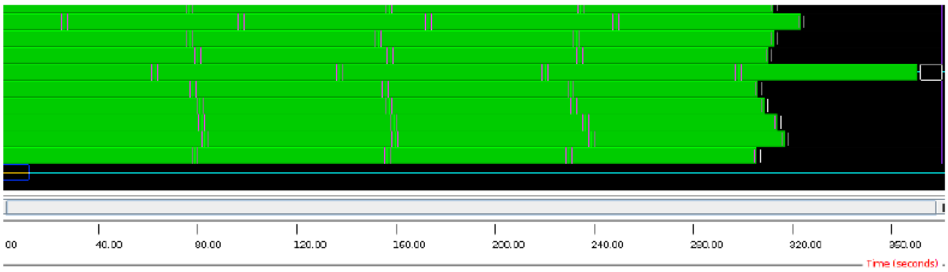


Figure 18. Jumpshot visualization of PIPS task transitions.

Each row represents one MPI process.
Green: PIPS task computation; **Bars:** Task transition

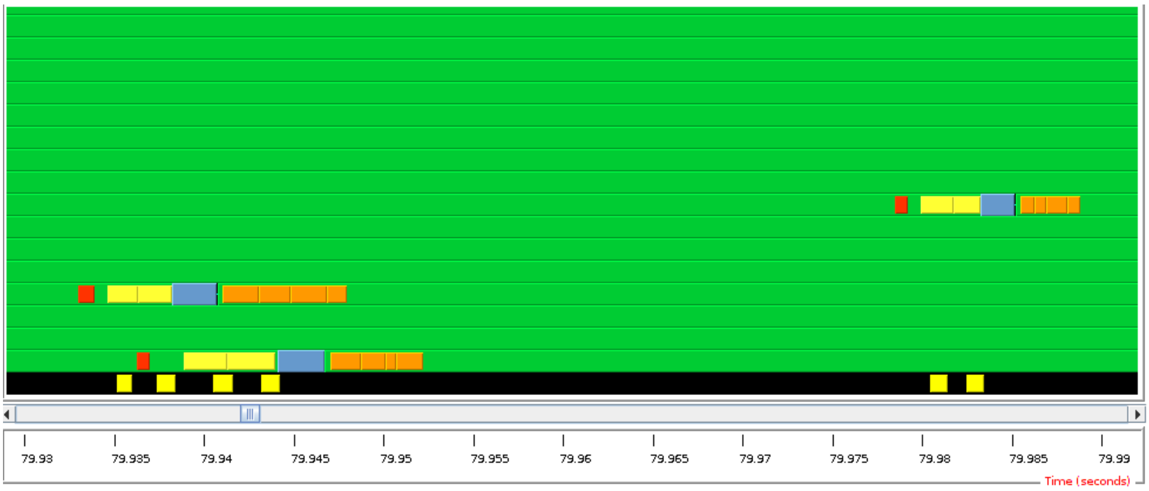


Figure 19. Jumpshot visualization for PIPS use of Turbine API.

This figure zooms in on representative task transitions.
Green: PIPS task computation; **Red:** Store variable; **Yellow:** Notification (via control task); **Blue:** Get next task; **Orange:** Retrieve variable; **Black:** Server process (handling of control task is highlighted in yellow)

PIPS usage of the Turbine API is shown in Figure 19. This figure exhibits multiple notable features. It shows that MPE can be used to inspect a distributed-memory, functional Von Neumann model, implemented by Turbine. Loads and stores are shown as each worker process transitions from one computational task to the next. Communication with the server, visible as the worker task put operation (yellow), is associated with the server handling of this operation (also yellow). Data-dependent operation is also illustrated by the handling of the store-notify-release cycle.

Numerical statistics regarding representative PIPS runs are as follows. For the 128-process case, 512 C++ function invocations were issued over a period of 333 s. Of that time, including the ramp-down at the end of the run, 37,934 s were spent in user code, for a total utilization of 88.5% worker becomes idle because of ramp-down, we obtain a utilization of 97.6%

In order to determine the ramp-down characteristic, the number of processes performing user computation over time was extracted from the MPE logs and accumulated. This result is plotted over time in Figure 20. As shown, the workflow begins running out of work after second 240, at which point the ramp-down begins.

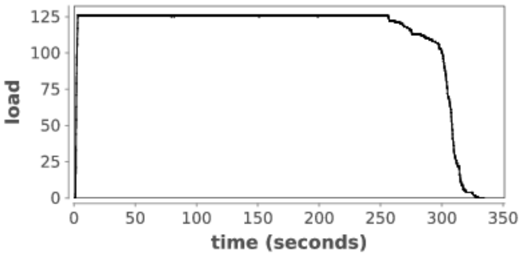


Figure 20. Computation load produced by PIPS run.

6.2.2. Discussion

The preceding PIPS discussion is a complete illustration of a typical Turbine application. Dataflow-based processing models, such as those developed to solve postprocessing problems from large scale numerical models of physical systems, may be elegantly expressed as Swift scripts, generating highly scalable Turbine programs. Such applications are capable of high concurrency and utilization, spending most of their time doing user work. These programs may be analyzed through their MPE logs, producing profile statistics and visual diagrams that illustrate program behavior.

7. Related Work

Our work is related to a broad range of previous work, including programming models and languages, task distribution systems, and distributed-data systems. These parallel task distribution frameworks and languages provide mechanisms to define, dispatch, and execute tasks over a distributed computing infrastructure or provide distributed access to data, but they have important differences from our work in scope or focus.

7.1. Many-task programming models

Recent years have seen a proliferation of programming models and languages for distributed computing.

One family comprises the "big data" languages and programming models strongly influenced by MapReduce [9], which are aimed at processing extremely large batches of data, typically in the form of records. Task throughput is not an important determinant of performance, unlike in our work, because very large numbers of records are processed by each task. The other major family closest to our work is Partitioned Global Address Space (PGAS) languages, which are designed for HPC applications and provide some related features but differ greatly in their design and focus.

Skywriting [25] is a coordination language that can distribute computations expressed as iterative and recursive functions. It is dynamically typed, and it offers limited data mapping mechanisms through a static file referencing; our language model offers a wider range of static types with a rich variable-data association through its dynamic mapping mechanism. The underlying execution engine, called CIEL [26], is based on a master/worker computation paradigm where workers can spawn new tasks and report back to the master. A limited form of distributed execution is supported, where the control script can be evaluated in parallel on different cluster nodes. The CIEL implementation is not designed specifically for high task rates: task management is centralized, and communication uses HTTP over TCP/IP rather than a higher-performance alternative.

Interpreted languages building on MapReduce to provide higher-level programming models include Sawzall [29], Pig Latin [27], and Hive [35]. These languages share our goal of providing a programming tool for the specification and execution of large parallel computations on large quantities of data and facilitating the utilization of large distributed resources. However, the MapReduce programming model just supports key-value pairs as input or output datasets and offers two types of computation functions, map and reduce, with more complex dataflow patterns requiring multiple MapReduce stages. In contrast, we implement a full programming language with a type system, complex data structures, and arbitrary computational procedures. Some systems such as Spark [44] and Twister [13] are incremental extensions to the MapReduce model to support iterative computions, but this approach does not give the same flexibility or expressiveness as a new language.

Dryad [17] is an infrastructure for running data-parallel programs on a parallel or distributed system, with functionality roughly a superset of MapReduce. In addition to the MapReduce communication pattern, many other dataflow patterns can be specified. Communication between Dryad operators can use TCP pipes and shared-memory FIFOs to be the communication as well as files. Dryad dataflow graphs are explicitly developed by the programmer; whereas our dataflow model is implicit. Furthermore, Dryad's dataflow graphs cannot easily express data-driven control flow, a key feature of Swift. Higher-level languages have been built on Dryad: a scripting language called Nebula, which does not seem to be in current use, and DryadLINQ [43], which generates Dryad computations from the LINQ extensions to C#.

Makeflow [42] is a parallel and distributed implementation of the popular Make utility. With a Make-like syntax, Makeflow supports parallel execution of data-driven workflows on distributed computing infrastructures. The Makeflow implementation architecture combines an "execution machine" for evaluation and job creation with a "submit machine" for interfacing with job schedulers. Current versions of Makeflow are interfaced with job schedulers such as SGE and distributed filesystems such as HDFS.

7.2. Approaches to task distribution

Scioto [12] is a lightweight framework for providing task management on distributed-memory machines under one-sided and global-view parallel programming models. Scioto has strong similarities to ADLB: both are libraries providing dynamic load balancing for HPC applications. Scioto uses a randomized work stealing algorithm, while ADLB provides some additional features such as task priorities and the ability to target processes with tasks.

Falkon [30] performs dynamic task distribution and load balancing. Task distribution is distributed but uses a hierarchical tree structure, with tasks inserted into the system from the top of the tree, in contrast to the flatter, decentralized structure of ADLB.

DAGuE [7] is a framework for scheduling and management of tasks on distributed and many-core computing environments. The programming model is minimalist, based around explicit specification task of DAGs, which does not bear much resemblance to a traditional programming languages and does not support data structures such as arrays.

Cilk-NOW [6] is a distributed-memory version of a functional subset of the Cilk task-parallel programming model. It provides fault tolerance and load balancing on networks of commodity machines but does not provide globally visible data structures, which makes it unable to carry out Swift-like semantics.

7.3. Approaches to distributed data

Linda [1] introduced the idea of a distributed tuple space, a key concept in ADLB and Turbine. This idea was further developed for distributed computing in Comet [20] which has a goal similar to that of our work but focuses on distributed computing and is not concerned with scalability on HPC systems. Turbine's data store is different in functionality because it is designed primarily to support the implementation of a higher-level language. It does not support lookup based on templates or approximate key, only lookup of values by exact key.

Recently, considerable research has been devoted to distributed *key-value stores*, which are not fundamentally different from tuple spaces but tend to emphasize simple data-storage rather than data-driven coordination and support simpler query methods such as exact key lookups or range queries. Memcached [15] is a simple RAM-based key-value store that provides high performance with no durability and minimal consistency guarantees; it provides a single, completely flat hash table with opaque keys and values. Redis [31] provides similar functionality to memcached, as well as a range of data structures including hashtables and lists and the ability to subscribe to data items. Other, more sophisticated key-value stores that are highly scalable, use disk storage, and provide consistency and durability guarantees include Dynamo [10] and Cassandra [19]. While these key-value systems provide a range of options for data storage, they do not take advantage of high-performance message passing available on many clusters and do not provide all of the coordination primitives that were required in Turbine to implement a dataflow language like Swift.

8. Conclusion

We have described three main contributions of the Turbine engine and its programming model.

First, we have identified many-task, dataflow programs as a highly useful model for many real-world applications, many of which are currently running in Swift. We provided projections of exascale parameters for these systems, resulting in requirements for a next-generation task generator.

Second, we identified the need for a distributed-memory system for the evaluation of the task-generating script. We identified the *distributed future store* as a key component and produced a high-performance implementation. This work involved the development of a data dependency processing engine, a scalable data store, and supporting libraries to provide highly scalable data structure and loop processing.

Third, we reported performance results from running the system on the SiCortex and Blue Gene/P. The results show that out system can achieve the required performance for extreme cases.

While the Turbine execution model is based on the semantics of Swift, it is actually much more general. We believe that it is additionally capable of executing the dataflow semantics of languages such as Dryad, CIEL, and PyDFlow and could thus serve as a prototype for a common execution model for these and similar languages.

Acknowledgments

This work was supported by the U.S. Department of Energy under the ASCR X-Stack program (contract DE-SC0005380) and contract DE-AC02-06CH11357. Computing resources were provided by the Mathematics and Computer Science Division and Argonne Leadership Computing Facility at Argonne National Laboratory. Work by Katz was 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 author(s) and do not necessarily reflect the views of the National Science Foundation.

We thank Matei Ripeanu and Gail Pieper for many helpful comments, and our application collaborators: Joshua Elliott (DSSAT); Andrey Rzhetsky (SciColSim); Miles Lubin, Cosmin Petra, and Victor Zavala (Power Grid); Yonas K. Demissie and Eugene Yan (SWAT); and Marc Parisien (ModFTDock).

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.

[2] T. G. Armstrong, J. Wozniak, M. Wilde, K. Maheshwari, D. S. Katz, M. Ripeanu, E. Lusk, and I. Foster. ExM: High level dataflow programming for extreme scale systems. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar), poster*, Berkeley, CA, June 2012. ACM, USENIX Association.

[3] 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 *Proc. Workshop on Many-Task Computing on Grids and Supercomputers, 2011*, 2010.

[4] ASCAC Subcommittee on Exascale Computing. The opportunities and challenges of exascale computing, 2010. U.S. Dept. of Energy report.

[5] 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, 1996. USENIX Association.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proc. of Annual Conf. on USENIX*, page 10, Berkeley, CA, USA, 1997. USENIX Association.

[7] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *Proc. Intl. Parallel and Distributed Processing Symp.*, 2011.

[8] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[10] 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.*, 41:205–220, Oct. 2007.

[11] E. Deelman, T. Kosar, C. Kesselman, and M. Livny. What makes workflows work in an opportunistic environment? *Concurrency and Computation: Practice and Experience*, 18:1187–1199, 2006.

[12] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. *Intl. Conf. on Parallel Processing*, pages 586–593, 2008.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *Proc. of 19th ACM Intl. Symp. on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, 2010. ACM.

[14] J. Evans and A. Rzhetsky. Machine science. *Science*, 329(5990):399–400, 2010.

[15] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004:5–, August 2004.

[16] 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.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

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

[19] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[20] Z. Li and M. Parashar. Comet: A scalable coordination space for decentralized distributed environments. In *2nd Intl. Work. on Hot Topics in Peer-to-Peer Systems, HOT-P2P 2005*, pages 104–111, 2005.

[21] M. Lubin, C. G. Petra, and M. Anitescu. The parallel solution of dense saddle-point linear systems arising in stochastic programming. *Optimization Methods and Software*, 27(4–5):845–864, 2012.

[22] M. Lubin, C. G. Petra, M. Anitescu, and V. Zavala. Scalable stochastic optimization of complex energy systems. In *Proc. SC*, 2011.

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

[24] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proc. HotPar*, 2010.

[25] D. G. Murray and S. Hand. Scripting the cloud with Skywriting. In *HotCloud '10: Proc. of 2nd USENIX Work. on Hot Topics in Cloud Computing*, Boston, MA, USA, June 2010. USENIX.

[26] 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.

[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proc. of 2008 ACM SIGMOD Intl. Conf. on Management of Data*, SIGMOD '08, pages 1099–1110, New York, 2008. ACM.

[28] C. Petra and M. Anitescu. A preconditioning technique for schur complement systems arising in stochastic optimization. *Computational Optimization and Applications*, 52:315–344, 2012.

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

[30] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proc. of 2008 ACM/IEEE Conf. on Supercomputing*, SC '08, pages 22:1–22:12, Piscataway, NJ, 2008. IEEE Press.

[31] Redis. `http://redis.io/`.

[32] J. Shalf, J. Morrison, and S. Dosanj. Exascale computing technology challenges. VECPAR'2010, 2010.

[33] H. Simon, T. Zacharia, and R. Stevens. Modeling and simulation at the exascale for energy and the environment, 2007. Report on the Advanced Scientific Computing Research Town Hall Meetings on Simulation and Modeling at the Exascale for Energy, Ecological Sustainability and Global Security (E3).

[34] R. Stevens and A. White. Architectures and technology for extreme scale computing, 2009. U.S. Dept. of Energy report, available at `http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Arch_tech_grand_challenges_report.pdf`.

[35] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2:1626–1629, August 2009.

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

[37] E. Walker, W. Xu, and V. Chandar. Composing and executing parallel data-flow graphs with shell pipes. In *Work. on Workflows in Support of Large-Scale Science at SC'09*, 2009.

[38] B. B. Welch, K. Jones, and J. Hobbs. *Practical programming in Tcl and Tk.* Prentice Hall, 4th edition, 2003.

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

[40] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37:633–652, 2011.

[41] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Scalable data flow programming for many-task applications. In *Proc. CCGrid*, 2013.

[42] 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*, 13:243–256, 2010.

[43] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of Symp. on Operating System Design and Implementation (OSDI)*, December 2008.

[44] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.