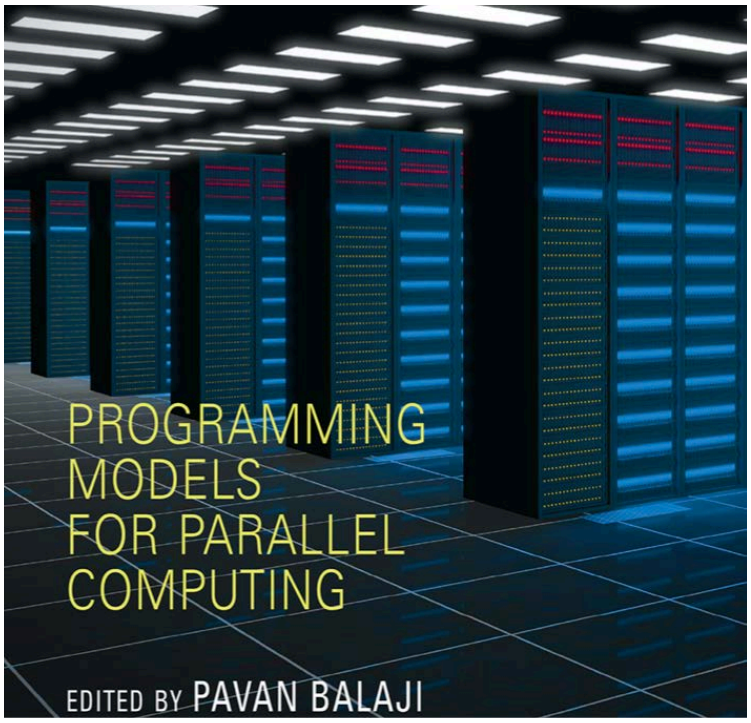

Chapter 10 Swift: Extreme-scale, Implicitly Parallel Scripting

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in LaTeX by the authors and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

To be provided by MIT Press

10 9 8 7 6 5 4 3 2 1

10 Swift: Extreme-scale, Implicitly Parallel Scripting

Timothy Armstrong, The University of Chicago
Justin M. Wozniak, Argonne National Laboratory and The University of Chicago
Michael Wilde, Argonne National Laboratory and The University of Chicago
Ian T. Foster, Argonne National Laboratory and The University of Chicago

Scientists, engineers, and data analysts frequently find themselves needing to execute a set of application tasks hundreds—or even millions—of times, for example to optimize a design simulation or to process large collections of data records. Such activities can be intellectually and administratively arduous, due to the need to orchestrate many data movements and application execution tasks, and to track the resulting outputs, which themselves often serve as inputs to further applications. Further complicating these activities is the frequent need to leverage distributed and parallel computing resources in order to complete computations in a timely manner.

Task-parallel programming models allow existing code (libraries or programs) to be rapidly developed into scalable applications. However, they generally do not capture the high-level *workflow structure* of the overall application. Concepts like iteration, recursion, and reduction are lost if the user must coordinate tasks with the task-parallel library. It is difficult to compactly express these abstractions in the event-handling style required by the master-worker model. Additionally, data management is lost, and data dependencies must be encoded in an ad hoc manner.

The Swift parallel scripting language represents a unique approach to this problem. Swift transparently *generates* an task-parallel ADLB program (cf. Chapter 8) from a high-level script, which contains data definitions, data dependencies, and links to external native code (i.e., C/C++/Fortran). This program can then be run on an MPI-based high-performance computer.

Like other scripting languages, Swift allows programmers to express computations via the linking together of existing application code by, for example, specifying that the output of program A be provided as input to tasks B and C, the outputs of which are then consumed by task D. This approach has the advantages of allowing for rapid application development and avoiding the need to modify existing programs. Swift supports concurrency implicitly, so that in our example, if tasks B and C have no other dependencies, they can both execute in parallel as soon as A completes. As described in the following, Swift is not limited to directed acyclic graph (DAG) dependency expressions.

Additionally, Swift introduces a powerful data model that allows for typical scalars (integers, floats, strings), arrays, structs, and so on. Swift also supports an unformatted byte array (called a *blob* for "binary large object"), which can hold arbitrary native data for messaging from one task to the next. Furthermore, Swift represents external files as variables, which can also be the subject of data dependent operation (similar to Makefiles). These features together can reduce greatly the costs of developing and running computations such as those referred to above.

In this chapter, we introduce the Swift programming model and execution model. We aim to provide enough information to allow the reader to write a Swift program. We first use simple examples to introduce key Swift concepts and then introduce the language syntax, demonstrating its broad applicability to highly productive large scale computation. We finally describe the distributed architecture that is used to run applications on even the largest parallel computers.

10.1 A First Example: Parallel Factorizations

We use a simple example to introduce the Swift language, computing the factors of all numbers, up to N, in parallel, and then produce a histogram of the popularity of each factor.

Swift script file: **factors.swift**

```
 1  int N = parseInt(argv("N"));
 2  bag<int> M[];
 3  foreach i in [1:N] {
 4    int factors[] = factorization(i);
 5    foreach f in factors {
 6      M[f] += 1;
 7    }
 8  }
 9
10  foreach b,i in M {
11    printf("%i: %i", i, bagSize(b));
12  }
```

Swift usage in the shell:

```
> swift-t factors.swift -N=10 | sort -n
1: 10
2: 5
3: 3
4: 2
5: 2
6: 1
...
10: 1
```

Figure 10.1: Swift example: Factorization

The Swift script is shown at the top of Figure 10.1 as `factors.swift`. This Swift program has one link to an external function, `factorization()` (line 4), which could be implemented in native code. This function returns all factors of a given integer, e.g., `factorization(12)`$\rightarrow$`1,2,3,4,6,12.`

The program begins by obtaining `N` from the user (line 1), then looping (line 3) from 1 to `N` *concurrently*. Swift internally uses the Asynchronous Dynamic Load Balancing (ADLB) model (Chapter 8) for task management. In this example, each loop iteration is implemented as an ADLB task, executing somewhere in the system.

Each `factorization(i)` call then executes as a task (line 4), returning the array of factors. Each factor must increment its count. This count is maintained in the *bag* for that factor, i.e., `M[f]` is incremented each time `f` occurred as a factor (line 6). `M[]` is thus defined as an array of bags, each containing integers (line 2). This structure should be recognizable as a MapReduce pattern; in fact, Swift can elegantly represent MapReduce [93] and various of its generalizations [7, 103].

Swift execution is shown at the bottom of Figure 10.1. For `N=10`, the factor 1 appears 10 times, 2 appears 5 times (once for each even value of `i`), and so on. The output is piped through `sort` because the `printf()` statements (line 11) execute in load-balanced, system-defined order.

10.2 A Real-World Example: Crystal Coordinate Transformation

We use a second example to show how the Swift language can be used to analyze data: in this case, to apply an data transformation to a 3D dataset. This example builds on the concepts in the previous example, with only slight additional complexity.

This scientific use-case is from X-ray scattering at the Advanced Photon Source at Argonne National Laboratory. The task is to perform a coordinated transform on a three-dimensional pixel array, converting the data from detector coordinates to real coordinates. Each chunk of the input data contributes to some set of output chunks, but the precise mapping is not known in advance. Thus, the transform function returns the list of output chunk identifiers as part of its output. Figure 10.2 shows an example use in two dimensions. The diamond-hashed chunks (input chunks 3, 6, 8, 12, and 13) contribute to output chunk 2. The transformed data from each is put in a bag and then merged into the output chunk. This application also clearly has MapReduce-like behavior.

This pattern is represented in the Swift script in Figure 10.3. The program has four external functions, implemented in C++, each prefixed with `cctw_`. (The C++ version is runnable as a stand-alone program, parallelized for a multicore machine with the Qt Concurrent library. Swift enables this same code to run across multiple nodes.) For each input chunk `i`, the input HDF file is read (line 3), obtaining the hyperslab corresponding to chunk ID `i`: represented by a Swift blob. Then, the chunk is transformed (line 6), producing arrays of output chunks and output IDs. For each output pair `j` (line 7), the

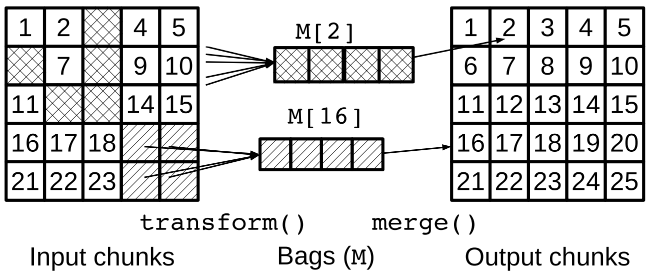

Figure 10.2: Crystal coordinate transformation dataflow pattern.

chunk is appended to the corresponding bag (line 9). Thus, in this application, each bag contains blobs (not integers as in the previous example).

Once each transform has completed, the blobs for each bag can be merged via a simple weighted addition (line 13). Then, the output chunks can be written to the corresponding HDF hyperslabs (line 14).

As in the factorization example, each call to an external function is run as concurrently as possible, limited only by data dependency. This approach allows Swift to make good use of massively parallel computers, without the direct use of lower-level libraries. Thus, we see how existing program components (C++ components, in this case) can be used to create a high-performance task-parallel computation through the use of a high-level script. Note also how Swift's bag, array, and blob features make it easy to distribute and compute over binary data on parallel computers.

10.3 History of Swift

The original implementation of Swift (called *Swift/K* because it is based on a runtime system called Karajan) was designed for coordination of large-scale distributed computations that make use of multiple autonomous computing resources distributed over varied administrative domains, such as clusters, clouds, and grids. Swift/K focused on reliability and interoperability with many systems at the expense of performance: execution of the program logic is confined to a single shared-memory *master* node, with calls to external executable applications dispatched to execution resources as parallel tasks over an execution provider such as Coasters [135] or Falkon [236]. Even in favorable circumstances with a fast execution provider executing tasks on a local cluster, at most 500–1000 tasks can be dispatched per second by Swift/K. This rate is insufficient for applications with more demanding performance needs such as a high degree of parallelism or short task duration.

```
bag<blob> M[];
foreach i in [1:n] {
  blob inputChunk = cctw_input("xray-data.hdf", i);
  blob outputChunks[];
  int outputIds[];
  (outputIds, outputChunks) = cctw_transform(i, b1);
  foreach chunk, j in outputChunks {
    int outputId = outputIds[j];
    M[outputId] += chunk;
  }
}
foreach g in M {
  blob b = cctw_merge(g);
  cctw_write(b);
}
```

Figure 10.3: Swift example: Crystal coordinate transformation

Optimizations to the language interpreter, network protocols, and other components could increase throughput, but a single-master architecture ultimately limits scaling and is unsuitable for applications with tasks with durations of hundreds of milliseconds or less or with a high degree of parallelism (more than several thousand parallel tasks) [225]. Thus, in order to address the needs of many demanding parallel applications, the current Swift implementation, sometimes called *Swift/T* (because it is based on a runtime system called Turbine [296], which uses ADLB), achieves high-performance by parallelizing and distributing script execution and task management across many nodes.

The current Swift language's syntax and semantics are derived from, and remain close to, the original Swift/K language. Swift focuses on enabling a *hierarchical* programming model for high-performance fine-grained task parallelism, orchestrating large-scale computations composed of external functions with in-memory data, computational kernels on GPUs and other accelerators [164], and parallel functions implemented in lower-level parallel programming models—typically threads or message-passing. These functions and kernels are integrated into the Swift language as typed *leaf functions* that encapsulate computationally intensive code, leaving parallel coordination, task distribution, and data dependency management to the Swift implementation.

Swift can be rigorously analyzed and enhanced by a range of compiler optimization techniques to achieve high efficiency and scalability for a broad range of applications on massively parallel distributed-memory computers. Its design is motivated by the limitations of current programming models for programming extreme-scale systems and addressing emerging problems such as programmability for nonexpert parallel programmers, abstraction of heterogeneous compute resources, and the composition of heterogeneous task types into unified applications.

We next provide an overview of Swift syntax and semantics. We then present the design and implementation of an efficient and scalable runtime system for this execution model, and techniques for efficiently compiling Swift for this style of runtime system, including compiler optimization techniques that allow applications developed in Swift to execute efficiently on massively parallel distributed-memory systems.

10.4 Swift Language and Programming Model

The main features that characterize Swift are:

- A **hierarchical programming model** where computationally intensive code is written in various other programming languages and parallel coordination is written in Swift.
- **Implicit parallelism** and relaxed execution ordering constraints: program statements can execute out-of-order, whenever input data is available.
- Control structures, including conditional if/switch statements and loop constructs, that are **semantically related to the equivalent imperative constructs**, but are adapted for implicit parallelism and monotonic data.
- Use of data types such as single-assignment variables with the property of *monotonicity*, which can ensure that results of computations are deterministic even with nondeterministic scheduling of tasks.

Swift can guarantee deterministic execution even with implicit parallelism because its standard data types are *monotonic*; that is, they cannot be mutated in such a way that information is lost or overwritten. A monotonic variable starts off empty, then incrementally accumulates information until it is *frozen*, whereupon it cannot be modified further. Programs that attempt to overwrite data will fail at runtime (or compile time, if the compiler determines that the write is definitely erroneous). If write operations that modify monotonic variables are *commutative*, then writes can be reordered without changing the final result.

If reads to Swift data types are constrained so that transient states are not observable, then we can achieve deterministic computation even with nondeterministic ordering of operations. Swift programs using futures-based data types with these restrictions on reads are deterministic by construction, up to the order of side-effects such as I/O. For example, the output value of an arbitrarily complex function involving many data and control structures is deterministic, but the order in which, say, print statements execute depends on the nondeterministic order in which tasks run. Further nondeterminism is introduced only by non-Swift code, such as the implementation of builtins (the library function `rand()`), and external functions (written in native code).

Basic Swift variables are single-assignment I-vars [216] (sometimes alternatively called futures), which are frozen when first assigned. All basic scalar primitives in Swift are semantically I-vars: ints, floats, booleans, and strings. Files can also be treated as I-vars, with an I-var in the language mirroring a file in the file system to which it is *mapped*. Assigning a mapped file variable in Swift then results in a file appearing at that path.

Composite data types can be incrementally assigned in parts but cannot be overwritten. The only composite data types that Swift originally supported were structs and associative arrays [292], both of which are monotonic futures-based data types. The *associative array* is the most complex and heavily used of the two. Integer indices are the default, but other index types including strings are supported. The array can be assigned all at once (e.g., `int A[] = f();`), or in parts (e.g., `int A[]; A[i] = a; A[j] = b;`). An array lookup operation on `A[i]` will return when `A[i]` is assigned. An incomplete array lookup does not prevent progress; other statements can execute concurrently.

The Swift language guarantees that variables are automatically *frozen* when the implementation is sure that no more writes will occur. This allows Swift code to refer directly to properties such as the size of arrays and ensures that reads of nonexistent array keys will eventually fail. The implementation of automatic freezing in Swift requires both compiler analysis and runtime support.

We introduce the Swift language here through a series of examples that illustrate its syntax and semantics. The examples use version 0.8.0 of Swift.[1]

10.4.1 Hello World

We begin with the Swift version of the classic "Hello, World" program in Figure 10.4, which needs two lines of code: the `import` statement that imports the builtin `io` module, then the call to the `printf` function from the `io` module to print a string.

```
import io;
printf("Hello World");
```

Figure 10.4: Swift example: Hello World.

Adding another `printf` in Figure 10.5 adds an interesting twist related to Swift's implicit parallelism (the following examples omit `import` statements to the Swift standard

[1]http://swift-lang.org/Swift-T

library). In Swift, the statements are allowed to run in any order because there is no data dependency between them: the program might print `Hello World` *after* `Goodbye World`.

```
printf("Hello World");
printf("Goodbye World");
```

Figure 10.5: Swift example: Hello/Goodbye World.

10.4.2 Variables and Scalar Data Types

Variables in Swift are strongly and statically typed: each variable's type is known at compile time and automatic conversion between types happens in few cases. The basic data types in Swift, which are treated as scalar values, are: `int` (64-bit integer), `float` (double-precision floating point), `string` (unicode string), `blob` (binary string), `boolean` (boolean value), `void` (no value: used for signaling), and `file` (file variable representing filesystem entry). Scalar variables are single-assignment I-vars: once a variable is declared, it can be assigned at most once; a second assignment leads to a runtime error. Figure 10.6 demonstrates various modes of declaration and assignment of variables.

```
// Declaration then assignment
int x;
x = 0;
printf(x);

// Combined declaration and assignment
float y = 2.0 + toFloat(x);

// Use before assignment is valid (dataflow is resolved at runtime)
string z;
printf(z);
z = "The quick brown fox jumped over the lazy dog";
```

Figure 10.6: Swift example: data types.

Variables can be assigned without being explicitly declared. If an variable name that has not previously been declared is assigned, Swift creates a new variable in the current scope with a type matching the expression on the right hand side of the assignment. This technique can be used in many but not all cases. For example, in Figure 10.7, automatic declaration can be used for `x` and `condition`, but `y` requires an explicit declaration because the assignments are both in inner scopes.

```
// x is automatically declared as a string variable
x = "Hello" + " " + " World";

// x is automatically declared as a boolean
condition = true;

if (condition) {
  y = x;
} else {
  y = "";
}
// Error! y is not defined in this scope
printf(y);
```

Figure 10.7: Swift example: Automatic declaration.

10.4.3 Dataflow Execution

As mentioned earlier, Swift is implicitly parallel, with program execution ordered by data dependencies. Thus, any two operators, function calls, or other parts of a Swift program can execute in parallel if there is no direct or indirect data dependency between them.

In Figure 10.8, the two calls to `f` can execute in parallel because neither depends on data produced by the other. The call to `g`, however, cannot execute in parallel with either `f` call because it depends on the data produced by both of them.

```
x = f(0);
y = f(1);
z = g(x, y);

printf("%i %i %i", x, y, z);
```

Figure 10.8: Swift example: dataflow parallelism between statements.

Different subexpressions of the same expression can also be evaluated in parallel. For example, Figure 10.9 implies the same pattern of parallelism as the previous example, despite the calls to `f` and `g` being embedded in the same expression.

```
printf("%i", g(f(0), f(1)));
```

Figure 10.9: Swift example: dataflow parallelism among expressions.

10.4.4 Conditional Statements

Conditional execution is supported by the `if` and `switch` statements. We omit discussion of `switch` statements here for the sake of brevity. The `if` statement's syntax is identical to that used in many imperative programming languages, such as C, but it executes in a data-dependent manner consistent with the rest of Swift. The condition of an `if` statement is evaluated in parallel with other statements in the enclosing block. Once the value of the condition is computed, the appropriate branch of the `if` statement is executed.

To illustrate how the `if` statement behaves in an implicitly parallel context, consider the code in Figure 10.10, which executes two computationally intensive simulation functions in parallel. After they finish, it compares the results and prints a message depending on the outcome. The programmer does not have to write code to explicitly synchronize and gather the results from the two parallel computations. Rather, the required synchronization happens automatically as part of the evaluation of the `if` statement condition, so that the message is printed once the outcome is known.

```
float f1, f2;

f1 = simulationA();
f2 = simulationB();

if (f1 > f2) {
  printf("Simulation A won!")
} else {
  printf("Simulation B won!")
}
```

Figure 10.10: Swift example: conditional execution with `if` statement.

10.4.5 Data-dependent Control Flow

The Swift `wait` statement and => chaining operator can be used to sequence statements by introducing explicit dependencies into a program. Either construct can be used to make a second statement depend explicitly on data produced by a first expression or statement, so that the second executes only after the data produced by the first is frozen, even if the second statement does not consume its value. This capability can used to add delays to a program, sequence messages reporting progress, or accommodate side effects in external functions. Note that, for these constructs to work, the statement that is to be waited on must produce some output. Most Swift functions have at least one output argument; if they do not, then it is straightforward to add a `void` output argument to signal when the function finishes executing. Figure 10.11 demonstrates the use of these features.

```
// Chaining of multiple statements
printf("Going to sleep") =>
  sleep(1) =>
  printf("Woke up") =>
  sleep(1) =>
  printf("Woke up again");

x = compute_something();

// The following forms are equivalent:
x => printf("Done!");

wait (x) {
  printf("Done!");
}
```

Figure 10.11: Swift example: data-dependent control flow.

The two constructs differ subtly in several ways. => waits on a statement, while `wait` waits on the expression supplied as its argument. Only statements that produce some kind of output variable support chaining. => can have any statement on its right hand side, while `wait` must be followed by a block enclosed in curly braces.

10.4.6 Foreach Loops and Arrays

Foreach loops are tied closely with Swift arrays, so we introduce both constructs simultaneously.

Arrays in Swift are associative arrays: finite maps of keys to values. The value type can be any Swift type. The default key type is `int` and other scalar key types such as strings are supported. Associative arrays with integer keys can also be viewed as *sparse* arrays: arrays with integer keys that do not need to be contiguous. There are multiple ways to declare and initialize arrays, as shown in Figure 10.12.

The workhorse control-flow construct in most Swift programs is the `foreach` loop for parallel iteration over members of Swift data structures, including arrays. Iterations of a foreach loop are independent and execute in parallel, provided that data dependencies allow. Iteration over an array constructed with the [*begin*:*end*:{*step*}] syntax is the idiomatic way to iterate over a range of integers. In general, this syntax instructs Swift to construct an array literal. However, when it is used in a loop iteration construct, Swift avoids construction of the intermediate array and thus the idiom comes with no perfor-

```
// Two equivalent ways of declaring an array A mapping integers to strings
string A[int];
string A[];

// Declaration of an array mapping strings to integers
int A2[string];

// Equivalent statements that initialize an array with the numbers from 1 to 4
B = [1, 2, 3, 4]; // List of values (keys 0-3 are implied)
B = [1:4]; // Integer range (keys 0-3 are implied)
B = [1:4:1]; // Integer range with explicit step of 1
B = { 0: 1, 1: 2, 2: 3, 3: 4 }; // Explicit keys
// Assigning piece-by-piece
B[0] = 1;
B[1] = 2;
B[2] = 3;
B[3] = 4;

// Declaring two-dimensional nested array
string C[][];
C[0][0] = "top-left";
C[0][1] = "top-right";
C[1][0] = "bottom-left";
C[1][1] = "bottom-right";
```

Figure 10.12: Swift example: array declarations.

mance penalty. To illustrate, Figure 10.13 shows code that builds an array by iterating over a range of integers and then iterating over the constructed array.

The loop iterations in Figure 10.13 may execute in any order and thus the results will likely not print in ascending order. In-order printing can be achieved by combining `for` loops (a construct distinct from `foreach`) with explicit data-dependent control flow (Section 10.4.5).

10.4.7 Swift Functions

So far we have only shown examples with Swift code at the top level of the program. Swift code can also be enclosed in functions for encapsulation and reuse. Swift functions must declare types and names of their input and output arguments. Functions return values by assigning the output arguments in the function body. Recursive function calls are allowed and tail recursion is supported in Swift: tail recursive calls of unlimited depth will not cause Swift to run out of stack space. Figure 10.14 illustrates Swift functions through different implementations of the factorial function.

```
// Get command-line argument n, default value of 100
int n = parseInt(argv("n", "100"));

float harmonic[];

// Compute the harmonic series.
// Note that this literally instructs Swift to construct an array containing
// integers 1 to n, then iterate over the constructed array. However, Swift/T
// always optimizes this to iterate over the range without building the array.
foreach i in [1:n] {
  harmonic[i] = 1 / toFloat(i);
}

// Iterate over values and indices
foreach x, i in harmonic {
  printf("H[%i] = %f", i, x);
}

printf("sum = %f", sum(harmonic));
```

Figure 10.13: Swift example: basic `foreach` loops.

Function bodies can begin executing as soon as the function is called, regardless of the state of their input and output arguments. In Figure 10.15, assignment of each function input is delayed by a different amount. The `printf` calls in the function will execute at approximately one-second intervals once inputs are assigned.

10.4.8 External Functions

Swift is designed as a language for parallel coordination and scripting: the performance-critical sequential computation work is typically outsourced to code written in other languages. Thus, Swift provides rich support for integration with external functions written in programming languages including C, C++, Fortran, Python, R, Julia, Tcl, alongside the command-line applications traditionally supported by Swift [292]. These functions can be called from Swift code by declaring an *external function* with Swift input and output argument types.

All external functions in native code (C/C++/Fortan) are called via bindings generated with SWIG [28]. The Swift compiler automatically generates all necessary code to manage data-dependent execution and marshal the input and output arguments.

Swift provides a high-level interface for Python, R, Tcl, and Julia, by providing builtin functions that call to the appropriate interpreter, which may be optionally linked with Swift at configure time [297]. These interpreters may, in turn, call language extensions written

```
x_val = parseInt(argv("x", "5"));

f1, f2 = fact2(x_val);

printf("fact(%i) = %i", x_val, f1);
printf("fact_tail(%i) = %i", x_val, f2);

// Recursive implementation of factorial.
(int result) fact(int x) {
  if (x == 0) {
    result = 1;
  } else {
    result = x * fact(x - 1);
  }
}

// Tail-recursive implementation of factorial.
(int result) fact_tail(int x, int accum) {
  if (x == 0) {
    result = accum;
  } else {
    result = fact_tail(x - 1, accum * x);
  }
}

// Compute factorial in two ways, illustrating multiple output arguments
(int r1, int r2) fact2(int x) {
  r1 = fact(x);
  r2 = fact_tail(x, 1);
}
```

Figure 10.14: Swift example: basic Swift functions computing factorials.

```
print_three(string x, string y, string z) {
  printf("%s", x);
  printf("%s", y);
  printf("%s", z);
}

a = "Now";
sleep(1) => b = "Later" =>
sleep(1) => c = "Even later";

print_three(a, b, c);
```

Figure 10.15: Swift example: delayed assignment of Swift function arguments illustrating execution of Swift function body before arguments are all assigned.

in native code, creating a powerful hierarchical programming model. Figure 10.16 shows a simple external function implemented in Tcl.

Swift also supports native-code parallel libraries written in MPI that accept a communicator on which to execute [298]. When provided with such a function, Swift dynamically creates a subcommunicator and runs the user code on it.

```
// Declaration of log to arbitrary base via Tcl
@pure @dispatch=WORKER
(float o) my_log (float x, float base) "mypkg" "0.1"
[ "set <<o>> [ expr log(<<x>>)/log(<<base>>) ]" ];

printf("log10(100) = " + my_log(100, 10));
```

Figure 10.16: Swift example: declaration of a external Tcl function. The Tcl fragment is the string literal between the square brackets. Swift variables `o`, `x`, and `base` are marshaled to and from Tcl with the angle bracket syntax. The syntax `"mypkg" "0.1"` loads the Tcl package `mypkg`, version 0.1, allowing additional Tcl libraries and/or extensions (native code libraries with Tcl bindings) to be referenced from the Tcl fragment.

The `@pure` function annotation, used in Figure 10.16, is used to assert that the function is deterministic and has no side-effects. This annotation allows the Swift optimizer to reuse results of the function instead of recomputing them, if needed. The function annotation, `@dispatch=WORKER`, tells Swift that the function may take a little while to run and should always be executed as an independent task. Other annotations are documented in the Swift user guide [263].

10.4.9 Files and App Functions

Swift supports files as a first-class data type that can be treated similarly to a scalar value in the program. It also supports *app functions*: command-line programs that are wrapped as typed Swift functions. Thus, scripts manipulating files and invoking command-line applications can be expressed with regular Swift variables and function calls, as shown in Figure 10.17. This feature means that Swift can be used to develop file-based workflows, as with Makefiles, with extreme scalability. For example, one user who wanted to test a C compiler under a wide range of tuning parameters used Swift to distribute runs over distributed-memory systems.

```
app (file out) cat (file inputs[]) {
  "/bin/cat" inputs @stdout=out
}

file inputs[] = glob("*.txt");
file joined <"joined.txt"> = cat(inputs);
```

Figure 10.17: Swift example: Concatenating all text files in a directory.

10.5 The Swift Execution Model

We briefly describe here Swift's execution model for *data-driven task parallelism.* The execution model provides a foundation for the semantics of the Swift programming language. An important property of the model is that, subject to reasonable constraints, the result of a computation in the execution model is deterministic even when tasks are executed in a nondeterministic order or in a concurrent manner with interleaved reads and writes while accessing a shared data store.

As mentioned earlier, Swift is runs on the Turbine distributed runtime system, which is well suited for massively parallel distributed systems. The core component of Turbine is ADLB; Turbine provides additional features to make it an attractive compiler target for the Swift compiler (STC), providing a small set of primitives that enable Swift to run.

In data-driven task parallelism, all computation is performed by *tasks*, which are abstracted as mathematical functions that take input values and compute outputs of various kinds. Once executing, tasks run to completion and are not preempted. Tasks communicate by reading and writing *shared data* that resides in a data store. A task declares a set of shared data items that it will read and the computed output of a task includes a set of write operations on shared data items. Shared data is also the main means of synchronization: execution of a task can be made dependent on shared data so that the task does not run until that data is available.

To visualize the execution model, we will use a graphical notation for *trace graphs* that show the tasks, shared data, and dependencies that arise during execution. A trace graph, such as Figure 10.18, illustrates a single runtime execution of a program. Note that a single static graph cannot always serve as a specification of the data-driven tasks program because dependencies emerge dynamically at runtime. Tasks may selectively read, write, or spawn based on values computed at runtime: the tasks and relationships between tasks may vary between different executions of the same program, e.g., if the input data is varied. A trace graph never has cycles: in the case of deadlocks, deadlocked tasks will have fewer in-edges than data dependencies.

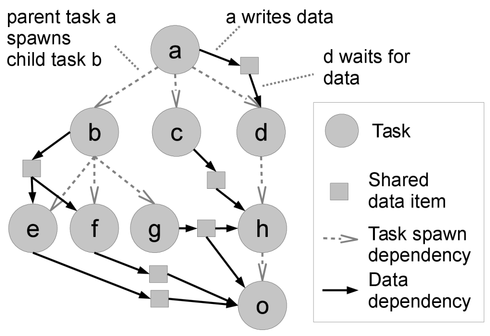

Figure 10.18: Trace graph showing task and data dependencies at runtime in data-driven task parallelism, forming a spawn tree rooted at task a. Data dependencies on shared data defer execution of tasks until the variables in question are frozen. Thus, for example, task h cannot execute until a data item is written by task c.

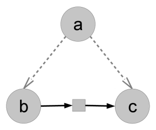

Figure 10.19: Task spawning two children that synchronize on an item of data.

Each task can spawn asynchronous *child tasks*, resulting in a *spawn tree* of tasks, as in Figure 10.18. In practice, tasks can be implemented through a set of parameterized *task definitions* that make up a program: at spawn time a task definition's parameters are bound to specific values by the parent to produce a child task. This allows parent tasks to pass data directly to their child tasks. For example, this could be small data such as numbers or short strings, along with references to arbitrary shared data. Shared data items can be read or written by any task that obtains a *reference* to the data. Shared data items provide for coordination between multiple tasks. For example, a task A can spawn two tasks, B and C, passing both a reference to a shared data item, which B writes and C reads, as shown in Figure 10.19. *Data dependencies*, which defer the execution of tasks, are the only way to synchronize between tasks. The execution model permits a task to write (or not write) any data it holds a reference to, allowing many runtime data dependency patterns beyond static task graphs.

Figures 10.20 and 10.21 use an example to illustrate how Swift may be translated into the execution model. The example application, an amalgam of several real scientific applications, runs an ensemble of simulations for many parameter combinations. The code

```
blob models[], res[][];
foreach m in [1:N_models] {
  models[m] = load(sprintf("model%i.data", m));
}

foreach i in [1:M] {
  foreach j in [1:N] {
    // initial quick evaluation of parameters
    p, m = evaluate(i, j);
    if (p > 0) {
      // run ensemble of simulations
      blob res2[];
      foreach k in [1:S] {
        res2[k] = simulate(models[m], i, j, k);
      }
      res[i][j] = summarize(res2);
    }
  }
}

// Summarize results to file
foreach i in [1:M] {
  file out<sprintf("output%i.txt", i)>;
  out = analyze(res[i]);
}
```

Figure 10.20: Swift code for the data-driven task trace graph of Figure 10.21.

(Figure 10.20) executes with implicit parallelism, ordered by data dependencies. Data dependencies are implied by reads and writes to scalar variables (e.g., `p` and `m`) and associative arrays (e.g., `models` and `res`). Swift semantics allow functions (e.g., `load`, `evaluate`, and `simulate`) to execute in parallel when execution resources are available and data dependencies are satisfied. This example illustrates the additional expressivity of the execution model over some common alternatives such as static task graphs or dataflow networks. Simulations are conditional on runtime values: data-driven task parallelism allows dynamic runtime decisions about what tasks to create. The task graph (Figure 10.21) shows an optimized translation to data-driven task parallelism. An unoptimized version would comprise more variables and tasks.

10.6 A Massively Parallel Runtime System

The Turbine runtime system enhances the ADLB load-balancing library, by supporting arbitrary user data, data dependencies, and miscellaneous builtin functions and other tools to support Swift.

Early prototype versions of Turbine extended ADLB with required functionality, such as a distributed data store and data-dependent task release [295]. Since then, we have further extended and enhanced Turbine to produce a complete and scalable distributed language runtime for Swift. This work includes task queue performance and scalability

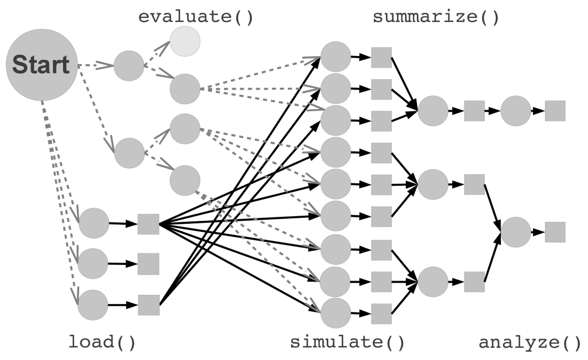

Figure 10.21: Visualization of optimized parallel tasks and data dependencies for the program of Figure 10.20, for parameters $M = 2$, $N = 2$, $S = 3$.

enhancements, work stealing to rebalance work between servers, richer data functionality, and support for garbage collection through reference counting.

10.7 Runtime Architecture

The Turbine/ADLB runtime is a distributed system that allows many *workers* to cooperate in executing massively parallel applications. It enables coordination between workers through three core services that are implemented efficiently and scalably: a *distributed data store* to store shared data, a *distributed task queue* to distribute work, and a *distributed dependency engine* that tracks data dependencies of tasks. Figure 10.22 illustrates the interactions between these services. These services provide operations that support distributed execution of Swift.

Task operations support adding and removing tasks from the distributed task queue. The $payload$ of each task is arbitrary binary data that can be interpreted in an application-dependent way. The task operations support adding tasks: enqueuing a task in the dependency engine or for immediate execution. "Get" operations remove a task from the queue of the desired $type$. Different Get variations support nonblocking gets of tasks useful when a worker can execute multiple tasks in parallel or when a programmer wants to overlap task execution with task gets.

A model of differentiated task types is used to support GeMTC GPU tasks [164], task dispatch to multiple worker types when integrated with the NAMD molecular dynamics software [230], and integration with Coasters for execution of remote command-line applications [135]. Parallel tasks have many applications, such as running ensembles of the OSUFlow particle tracing application [298]. The use of task priorities to prioritize criti-

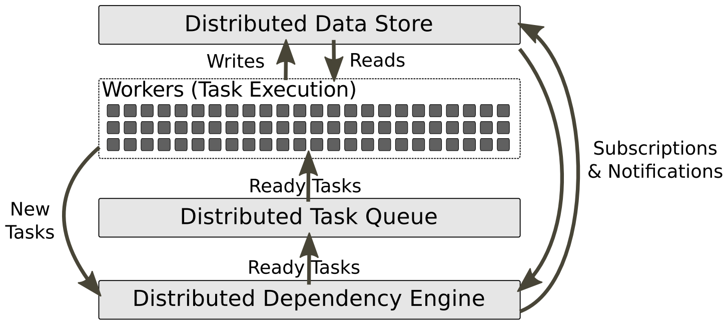

Figure 10.22: A view of the Swift distributed runtime (Turbine/ADLB) as distributed services enabling coordination between workers. Tasks created by code running on workers are passed to the dependency engine. The dependency engine holds tasks until required input data are available, and then passes the tasks to the task queue. Tasks are then sent from the task queue to workers to be executed. While executing, tasks can read and write the distributed data store. Writes to the distributed data store can trigger notifications to the dependency engine if the dependency engine has subscribed to that data.

cal tasks (e.g., tasks that are longer-running or on the critical path of the application) can significantly improve system utilization and reduce time-to-solution [14, 17]. Rank and node-level targeting—both `Hard` and `Soft`—have found applications in data-intensive applications where data is stored locally on the compute node and the cost of remotely reading data is significant [100, 299].

Runtime data operations allow creating, reading, writing, subscribing to, and reference counting of shared data items in the data store.

In order to implement the three distributed services provided by the runtime system, MPI processes are divided into two roles: *workers* and *servers*. Figure 10.23 shows a common way of distributing servers: one server per node. The system can be scaled up arbitrarily by proportionally adding processes of both types. Worker processes can execute any program logic, coordinating with each other using the data and task operations provided by distributed services. The distributed services are implemented by the server processes and accessed with remote procedure calls (RPCs).

Implementation of an efficient and scalable task queue hinges on two key features: efficient task-matching algorithms and data structures to maximize throughput per server, and scalable work-distribution algorithms to handle load imbalances between servers. Task matching only solves the problem of matching work on an individual server to that server's own workers. If a server runs out of work, then it must somehow acquire more work from another server to prevent its workers from sitting idle. Such load imbalances are common

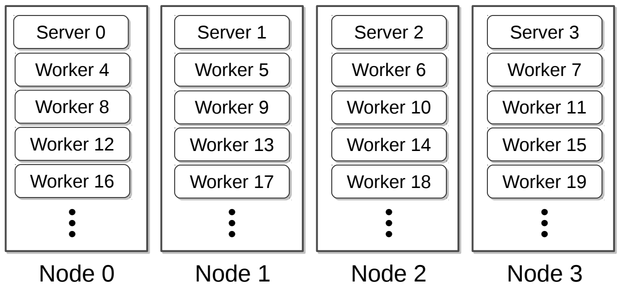

Figure 10.23: Runtime process layout on a distributed-memory system. Worker and server processes are mapped onto multicore systems.

in practice, and thus moving work from overloaded to underloaded servers efficiently is critical. Novel work stealing enhancements were used to address this problem [15].

The runtime's data store implements a distributed data store with semantics based on the abstract data store described below. Data store keys are 64-bit integers and the key space is partitioned between servers in a round-robin manner. Multiple placement strategies are possible when a worker calls `Create`. The current strategy used is to place the data on the nearest server, which improves data locality, but can lead to problems with load imbalance. Each data store key has an associated type tag. For compound data structures, additional type information about members is stored in various ways.

Garbage collection and automatic freezing are supported by a reference counting mechanism. Data-dependent task release is based on a key/path pair becoming frozen. Release is implemented efficiently through a subscription mechanism: any process in the system can subscribe to a key/path pair. Subscriptions are tracked by the server to which the key maps, and when the frozen state is entered, a notification message is set to the subscriber.

Efficient memory management is challenging in a distributed context, especially in the highly dynamic execution model of data-driven task parallelism, because references to a data item may be held by many processes at any given time. The classic memory management problem is generally formulated as the problem of detecting when no direct or indirect references to a data item are held by the executing program. The variable freezing problem can be formulated similarly: detecting when no `Write` references are held.

We tackle both problems with *automatic distributed reference counting*. We give each shared data item two reference counts (*refcount*), one for read references and one for write references. When a data item's write refcount drops to zero, it is frozen and cannot be written; when both refcounts drop to zero, the data can be deleted. Single-assignment variables do not require special treatment, but for variables such as arrays, where multiple

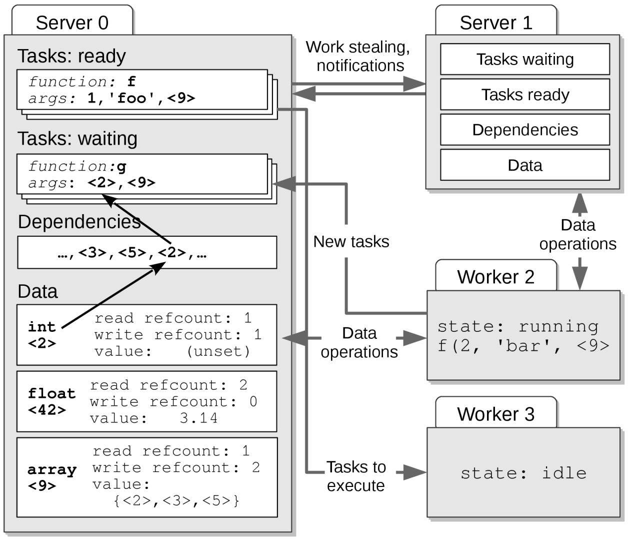

Figure 10.24: Runtime architecture showing distributed worker processes coordinating through task and data operations. Ready/waiting tasks and shared data items are stored on servers, with each server storing a subset of tasks and data. Servers must communicate to redistribute tasks through work stealing, and to request/receive notifications about data availability on other servers.

assignments are possible, refcounts must be correctly incremented and decremented to track the number of tasks and data structures with references to a key. A well-known weakness of reference counting is that it cannot handle cycles of references. The Swift data model does not permit such reference cycles, which avoids the problem.

10.8 Performance

Large-scale experiments were performed on the Blue Waters supercomputer [109] using Swift. Figure 10.25 shows our scalability and task throughput results obtained by running an embarrassingly parallel Swift program that exercises task matching and work stealing. Swift achieved a peak throughput of 1.47 billion tasks/s on 524,288 cores running the Sweep benchmark [16]. Tasks of 1 ms or more achieve high efficiency; the servers are lightly loaded and queuing delays are minimal.

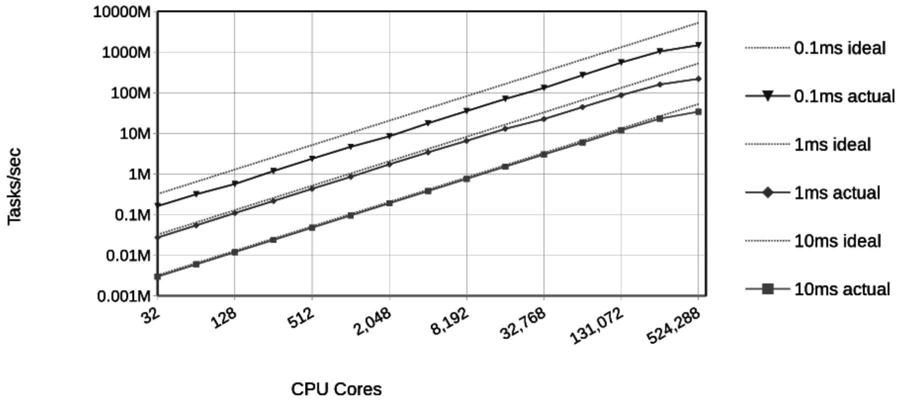

Figure 10.25: Throughput and scaling of runtime system for varying task durations.

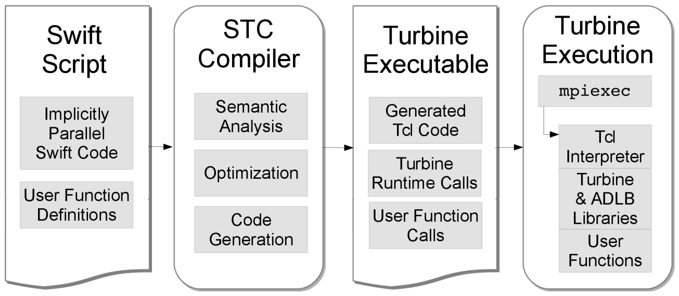

Figure 10.26: The STC compiler is in the middle of the Swift toolchain and translates high-level Swift code into execution code for the Turbine runtime.

10.9 Compiling Swift for Massive Parallelism

STC is a whole-program optimizing compiler for Swift that compiles high-level Swift code to run on the distributed runtime described in Section 10.6. STC generates code in the Tcl scripting language that can execute on the distributed runtime system; the runtime exposes ADLB and supporting libraries via Tcl bindings, enabling ease in the usage and debugging of the generated code. Figure 10.26 illustrates how STC fits into the Swift toolchain. STC implements optimizations aimed at reducing communication and synchronization without loss of parallelism. An intermediate representation captures the execution model, allowing optimization to reduce synchronization and runtime task/data operations involving shared

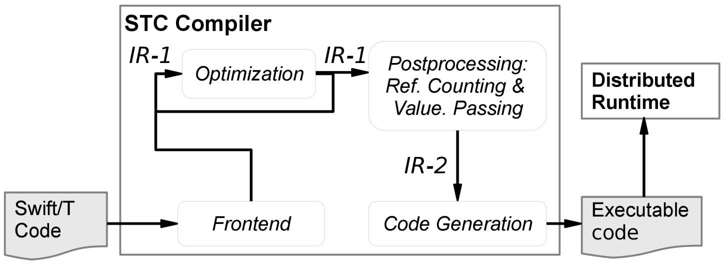

Figure 10.27: STC compiler architecture. The frontend produces IR-1, which is increasingly optimized by successive passes. Postprocessing adds intertask data passing and reference counting information to produce IR-2 for code generation.

data. It enables garbage collection by reference counting which is further optimized. We briefly survey the effectiveness of the compiler optimizations on several benchmarks. Further details of the compilation process can be found in an earlier paper [16] and Armstrong's Ph.D. dissertation [15].

To optimize a wide range of data-driven task parallelism patterns, we need compiler optimization techniques that can understand the semantics of task parallelism and monotonic variables in order to perform major transformations of the task structure of programs to reduce synchronization and communication at runtime, while preserving parallelism. Excessive runtime operations impair program efficiency because tasks waste time waiting for communication; they can also impair scalability by causing bottlenecks in the data store or task queue services.

The STC compiler uses a medium-level intermediate representation (IR) that captures the execution model of data-driven task parallelism. The tree structure of the intermediate representation can be mapped to the spawn tree of tasks, and dependencies through single-assignment data types are a first-class part of the IR. Two IR variants are used in STC, as shown in Figure 10.27. IR-1 is generated by the compiler frontend and then optimized. IR-2 includes additional information for code generation: explicit bookkeeping for reference counts and data passing to child tasks.

Detailed measurements and a comparison to related compilation and optimization work have been presented in prior publications [16, 15].

10.10 Related Work

Ousterhout [224] has written eloquently about the rationale and motivation for scripting languages, the difference between programming and scripting, and the place of each in the scheme of applying computers to solving problems.

Coordination languages such as Linda [8], Strand [110], and PCN [111] support the composition of implicitly-parallel functions programmed in specific languages and linked with the systems. In contrast, Swift coordinates the execution of what are typically legacy applications coded in various programming languages. Linda defines primitives for concurrent manipulation of tuples in a shared "tuple-space". Strand and PCN, like Swift, use single-assignment variables as their coordination mechanism. Linda, Strand, PCN and Swift are all dataflow-driven: processes execute only when data are available.

Swift has its origins in the Virtual Data Language (VDL) [112] developed within the Grid Physics Network (GriPhyN) project for management of large-scale data analysis computations. The term virtual data is intended to indicate that program rules define how data is to be produced; required computations are then performed when the user calls for the final data product.

Execution models and runtime systems combining task parallelism with data dependencies for HPC applications have been explored by several groups [104]. Tarragon [79] and DaGuE [49] implement efficient parallel execution of explicit dataflow DAGs of tasks from within an MPI program. ParalleX [152] provides a programming model through a C++ library that encompasses globally-addressable data and futures, with the ability to launch tasks based on dataflow. StarPU [20] and OmPSS [56] both provide lower-level library and pragma-based interfaces for executing tasks with data dependencies on CPUs and accelerators on distributed-memory clusters.

Habanero Java [264] and Habanero C [77] support asynchronous task parallelism with data dependencies on shared-memory nodes. Extensions to Habanero C support some inter-node parallelism with integration between MPI primitives and Habanero C, although this falls short of providing transparent task migration between nodes. X10 supports asynchronous task parallelism, but synchronization is based on a finish statement and termination detection algorithms, instead of data-dependencies [265].

The Asynchronous Dynamic Load Balancer (ADLB) [182], the basis of our runtime system, is highly scalable and has been successfully used by large-scale physics applications. However, its initial version did not support shared global data and the task queue performance was significantly extended through its integration into Swift.

Scioto (Chapter 9) 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 such as task priorities, work types, and targeted tasks.

Recent work on systems such as Sparrow [225], CloudKon [245], and Apollo [50] has attempted to improve throughput of task schedulers in cloud computing to enable workloads composed of "tiny tasks" on large clusters. These systems must deal with problems such as unreliability of workers and the need to enforce scheduling policies for shared resources. As a result of this and other implementation choices, they are unable to achieve anywhere near the efficiency of our runtime system: typical per-task overhead is tens to hundreds of milliseconds.

The MATRIX task scheduler [289], like Swift, seeks to implement high-performance distributed task scheduling with policies such as data-aware scheduling. However, MATRIX is built on a general-purpose key-value store. Our work shows that special-purpose data structures for task matching are required to achieve high performance given scheduling policies such as location-awareness and priorities.

10.11 Conclusion

We have described a hierarchical programming model for massively-parallel computing, which uses a high-level implicitly parallel language, Swift, to orchestrate computational tasks implemented in a range of other programming languages. This approach is a promising direction for addressing future systems challenges of unreliability and heterogeneity while making it substantially easier for nonexpert programmers to construct highly scalable parallel applications.

Swift's distributed Turbine runtime system faces many challenges, some of which—scalable synchronization, task matching, and task distribution—have been addressed with algorithms and data structures that address the problems of implicitly parallel dataflow-based programming models. These new approaches enabled great improvements in runtime performance and made the programming model performant and scalable enough to be attractive for many applications that fit the execution model.

Our experience with Swift provides strong evidence that that a combination of runtime algorithms and compiler techniques can enable high-level implicitly parallel code to drive fine-grained task-parallel execution at massive scales, rivaling the efficiency and scalability of hand-written parallel coordination code for common patterns of parallelism at scales from tens of cores to half a million cores and for a range of task-parallel application patterns including iterative optimization, tree search, and parallel reductions.

The system described in this chapter has been used for production science applications running on over 100,000 cores in production and over 500,000 cores in testing. Application of both compiler and runtime techniques was essential to reaching this scale. The Swift programming model offers a combination of ease of development and scalability that has proven valuable for developers who need to rapidly develop and scale-up applications, and do not have the time, expertise, or need to implement, optimize and debug applications in a lower-level distributed-memory programming model like MPI. Applying Swift to new and different problems will reveal further strengths, weaknesses, and opportunities.

Swift is an open source project with documentation, source code, and downloads for Swift/K and Swift/T available at `http://www.swift-lang.org`.

Acknowledgments

This research was supported in part by NSF grants OCI-721939 and OCI-0944332 and by the U.S. Department of Energy under contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility, XSEDE, Open Science Grid, the UChicago/Argonne Computation Institute's Beagle supercomputer, and the Amazon Web Services Education allocation program.

We gratefully acknowledge the contributions of current and former Swift team members, collaborators, and users: Glen Hocky, Hemant Sharma, Jun Park, Jon Almer, Ray Osborn, Guy Jennings, Jonathan Ozik, Sarah Kenny, Allan Espinosa, Zhao Zhang, Luiz Gadelha, David Kelly, Milena Nokolic, Jon Monette, Aashish Adhikari, Marc Parisien, Michael Andric, Steven Small, John Dennis, Mats Rynge, Michael Kubal, Tibi Stef-Praun, Xu Du, Yadu Nand Babuji, Ketan Maheshwari, Joshua Elliott, Zhengxiong Hou, and Xi Li. The initial implementation of Swift was the work of Yong Zhao and Mihael Hategan; Karajan was designed and implemented by Hategan under an effort led by Gregor von Laszewski. Swift/T is the work of Justin Wozniak and Timothy Armstrong, with contributions by Yadu Nand Babuji.