Skywriting on CIEL

Programming the Data Center

DEREK G. MURRAY AND STEVEN HAND

Derek G. Murray is currently completing his PhD at the University of Cambridge Computer Laboratory. Derek leads the CIEL project, which forms the basis of his thesis research into expressive programming models for distributed computation. At other points, his research interests have included OS virtualization, compiler optimization, and high-performance computing.
Derek.Murray@cl.cam.ac.uk

Steven Hand is a Reader in Computer Systems at the University of Cambridge Computer Laboratory. His interests span the areas of operating systems, networks, and security.
Steven.Hand@cl.cam.ac.uk

How do you write a program that runs across hundreds or thousands of computers? As data sources have proliferated, this question has spread beyond the domain of a few large search engines to become a concern for a variety of online services, large corporations, and academic researchers. This article introduces Skywriting and CIEL, which are, respectively, a programming language and a system designed to run a large class of algorithms on a commodity cluster.

Large-scale data processing requires parallelism, and useful parallelism requires *coordination* between processes. In a shared-memory system, coordination might involve updating a shared variable or signaling a condition variable; in a distributed system there is no shared memory, so processes communicate by sending *messages*. However, *explicit message passing* (using network sockets or a library such as MPI) is ill-suited to large commodity clusters, because it requires the programmer to specify the recipient host for every message. In these clusters, machines often go offline due to failure or planned maintenance, or they may be reassigned to another user. In general, cluster membership is far more dynamic than the supercomputers for which explicit message-passing libraries were first developed, and maintaining cluster membership information manually is a challenging distributed consensus problem.

The challenges of programming in a distributed system have led to the rise of *distributed execution engines*. These systems also send messages internally, but they virtualize the cluster resources beneath a high-level programming model. In 2004, Google announced MapReduce, which requires the implementation of just two functions—map() and reduce()—and frees the developer from having to implement parallel algorithms, distributed synchronization, task scheduling, or fault tolerance. Hadoop (an open-source implementation of MapReduce) was released soon afterwards, and has become widely used in many organizations, including Amazon, eBay, Facebook, and Twitter. In 2007, Microsoft published Dryad, which is a generalization of MapReduce that supports a broader class of algorithms, including relational-style queries with joins and multiple stages.

Most distributed execution engines divide computations into *tasks*, which are atomic and deterministic fragments of code that run on a single host. The power of an execution engine comes from its ability to track *dependencies* between tasks, and hence coordinate their execution and data flow. In increasing order of power, these dependency structures include:

1. **Bag of tasks.** In the simplest model, a job is divided into independent tasks, and terminates when all tasks have completed. For example, in SETI@home, the tasks are digitized chunks of radio transmissions, to which the same analyses are applied in parallel.
2. **Fixed dependencies.** There are task dependencies, but they are not controlled by the programmer. For example, in MapReduce, there are only two classes of task: map tasks and reduce tasks. By definition, reduce tasks consume the output of map tasks, so they must run after all map tasks have completed.
3. **Explicit dependencies.** The dependencies between tasks are programmer-controlled, and form an arbitrary directed acyclic graph (DAG). For example, in Dryad, a job is specified as a graph of vertices (representing the behavior of tasks) and channels (representing data flow between vertices).

Each successive model includes the previous one as a special case. Conversely, a more powerful system can be simulated using a *driver program* outside the cluster that submits multiple jobs to a less powerful system. Nevertheless, the advantage of a more powerful system is that the whole job enjoys the benefits of running on an execution engine, especially fault tolerance. The problem is that the most powerful model—explicit dependencies—requires all tasks and dependencies to be declared in advance, which limits the set of algorithms it can represent.

Many algorithms in machine learning, graph theory, and linear algebra are *iterative*, which means that they loop—performing some work in parallel—until they reach a fixed point. This means that the number of tasks cannot be known in advance. To support these algorithms on an execution engine, we introduce *dynamic dependencies*:

4. **Dynamic dependencies.** The dependencies between tasks are programmer-controlled and form an arbitrary DAG. However, each task may choose either to *publish* its outputs or *spawn* child tasks and *delegate* production of its outputs to one or more of its children.

The dynamic dependencies model supports iterative algorithms: one task implements the convergence test and either produces the current result or spawns another iteration. It also trivially supports all of the less powerful models. However, this raises the problem of how to specify a job that includes dynamic dependencies.

The Skywriting Scripting Language

To solve this problem, we created Skywriting, a scripting language for specifying dynamic dependency graphs. The language has three defining features:

- Skywriting is a "full" programming language (i.e., it is Turing-complete).
- A Skywriting script can spawn parallel tasks at any point in its execution.
- Any valid Skywriting script can be transformed into a dynamic dependency graph.

As a result, Skywriting is well suited to specifying jobs that run on CIEL (see the next section). Skywriting is interpreted, dynamically typed, and based on a C-like syntax. The language includes imperative control-flow constructs such as `while` loops and `if` statements, but it also includes first-class functions, which enables a functional style of programming. Figure 1 (next page) shows an implementation of a parallel Fibonacci algorithm, which illustrates the main syntactic features of the language (although the algorithm used is extremely inefficient).

```
function fib (n) {
   if (n <= 1) {
       return n;
   } else {
       x = spawn(fib, [n - 1]);
       y = spawn(fib, [n - 2]);
       return *x + *y;
   }
}

return fib(10);
```

Figure 1: Complete Skywriting script for computing the 10th Fibonacci number

The spawn() function creates a new parallel task. The first argument is a callable object: in the example, it is the name of a function (`fib`), but it could alternatively be an anonymous function or lambda expression. The second argument is a list of arguments that will be passed to the function in the new task. The return value from spawn() is a *future*, which can be passed to other invocations of spawn() in order to build a task dependency graph.

The `return` statement publishes the given expression as a task output. At the top level, a `return` statement publishes the overall job output. In a spawned function, a `return` statement publishes the current task's output. However, as you might expect, a `return` statement within a called function simply returns the value of the given expression to the caller.

Applying the * (dereference) operator to a future causes the current task to block until the producing task has returned the relevant value. In Figure 1, both `x` and `y` are futures, so the expression (`*x + *y`) will block the current task until both values are available. Although a task logically "blocks" when it dereferences a future, the runtime system actually creates a new task, which is called a *continuation task*. The current task delegates its output to the continuation task and adds dependencies on the dereferenced values. Therefore, the *-operator in Skywriting helps the programmer to build jobs with data-dependent control flow, without having to construct the dynamic dependencies manually.

Handling Larger Data

As an interpreted language, Skywriting is well suited to building coarse-grained task graphs, but less ideal for compute- or I/O-intensive work. Therefore the language includes mechanisms for handling large objects and executing external code. Figure 2 shows the implementation of a simple script that invokes external code to count the words in three Web pages.

The ref() function creates a *reference* to the given URL. Like a pointer in C, a reference is a handle to a potentially large object, and references may be exchanged efficiently between tasks. In Figure 2, the references refer to publicly accessible HTTP servers, but it is more common to use the `ciel://` URL scheme to refer to objects stored in the cluster (see the next section).

The exec() function synchronously invokes some external code. (There is also a spawn_exec() function which creates a task to invoke some external code asynchronously and takes the same arguments as exec().) The first argument is the name of an *executor*, which is effectively a loader for the invoked code. In Figure 2, the executor is `stdinout`, which is used to run legacy UNIX utilities that communicate using

standard input and output files. Other executors exist for Java and .NET classes. The second argument is the args dictionary, which contains executor-specific arguments: `inputs` is a list of references that will be concatenated and piped into standard input; `command_line` is the `argv` array for the process to be executed. The third argument controls the number of outputs: exec() returns a list containing one reference for each output. Note that calls to exec() are typically wrapped in a library function (e.g., stdinout(), java()) that sets the arguments appropriately.

```
news_sites = [ref("http://www.bbc.co.uk/news/"),
              ref("http://www.cnn.com/"),
              ref("http://www.foxnews.com/")];

function word_count (doc) {
   result = exec("stdinout",
                 {"inputs" : [doc], "command_line" : ["wc", "-w"]}, 1)[0];
   return *result;
}

total = 0;
for (site in news_sites) {
   total += word_count(site);
}
return total;
```

Figure 2: Complete Skywriting script for invoking `wc` on three Web pages

Since the aim of Skywriting is to support data-dependent control flow, it must be possible for a script to interrogate the output of a call to exec(). Therefore, the * operator can also be applied to a reference, which has the effect of loading the value into the script context. In Figure 2 the * operator is applied to the result of `` `wc -w` ``, which is an ASCII-encoded integer. Skywriting expects that a dereferenced reference is a valid value in JavaScript Object Notation (JSON). JSON resembles the syntax of Skywriting, and JSON parsers and generators exist for most popular languages.

CIEL: A Universal Execution Engine

To run Skywriting scripts on a cluster, we needed an execution engine that can handle dynamic dependencies: therefore we also developed CIEL, which is a *universal* execution engine. CIEL is universal in two senses: informally, because its execution model supports all existing task-parallel execution models, and formally, because the language for specifying the coordination between tasks (i.e., Skywriting) is Turing-complete.

Like many other execution engines, CIEL uses a master-worker architecture (Figure 3, next page). The central *master* schedules tasks for execution: it is similar to the JobTracker in Hadoop or the Job Manager in Dryad. The master stores metadata about the tasks and objects in the system and the (potentially dynamic) dependencies between them. The *scheduler* identifies when tasks become runnable and chooses where to run them, using a policy that reduces the amount of data copied across the network.

A CIEL cluster also includes several *workers*, which execute tasks and store data objects. When a task arrives at a worker, it is dispatched to the relevant executor, which corresponds to the notion of an executor in the Skywriting exec() and spawn_exec() functions. The executor retrieves the task's dependencies and makes them available to the invoked code in an appropriate manner. For example,

the `stdinout` executor forks a process to run a given command-line and writes the dependencies in turn to standard input; the Java executor exposes the dependencies as `InputStream` objects.

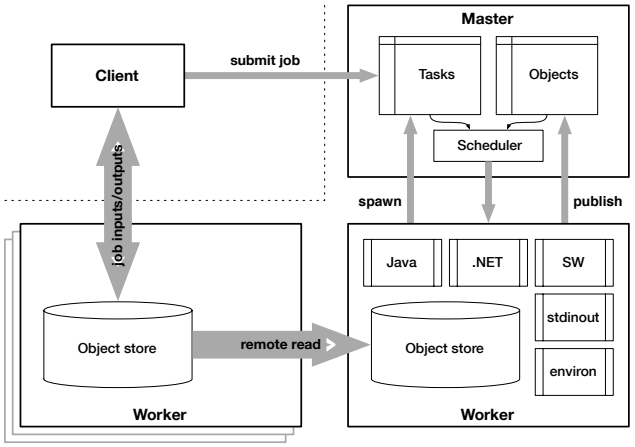

Figure 3: A CIEL cluster has one master and several workers. Arrows indicate communication between components; thicker arrows represent higher-bandwidth transfers.

Each worker also has an *object store*, which is backed by disk storage. CIEL uses objects to represent the input data, intermediate data, and results of each job. An object can contain arbitrary binary data, or it may be more structured (e.g., JSON format). Together, the master and the object stores form a simple distributed file system: every object has a unique name, but identical objects on different machines can share the same name, which enables replication. CIEL includes tools for transferring data into and out of a cluster, with support for replication and partitioning. Once loaded into the cluster, an object may be referenced from a Skywriting script using the ref() function and the `ciel://` URL scheme (Figure 4). A common pattern involves partitioning a file into several objects, then storing an *index object* that contains a list of references to the partitions. This is useful for specifying MapReduce-style jobs over large input data with many partitions.

```
ciel://[<HOSTNAME>:<PORT>]/<OBJECT_ID>
```

Figure 4: Syntax of a `ciel://` URL. If the hostname and port are omitted or become stale, the local master is consulted for up-to-date location information.

The main advantage of supporting dynamic dependency graphs in the execution engine is that CIEL can provide transparent fault tolerance across a whole job, including the data-dependent steps. CIEL provides fault tolerance for the client, workers, and master. Client fault tolerance is trivial, since there is no driver program running on the client, and so it plays no further part once it has submitted a job (except to collect the results). Worker fault tolerance is achieved by re-executing any tasks currently running on a failed worker, and recursively re-executing tasks to recreate any missing intermediate data. CIEL supports master fault tolerance by persistently logging messages from the workers, including spawned tasks and produced outputs: when the master comes back online, it can reconstruct its internal state by replaying the log.

Conclusion

We originally developed CIEL as a successor to systems like MapReduce, Hadoop, and Dryad, which run on large commodity clusters. Our performance evaluation, which is presented in a paper at NSDI '11, demonstrates that CIEL can achieve performance equal to or better than a less expressive system such as Hadoop. Performing control flow within the cluster leads to better performance and fault tolerance, by removing the need for a driver program outside the cluster.

We are now exploring other parts of the design space for dynamic dependency graphs. Skywriting and CIEL are suitable for coarse-grained parallelism on loosely coupled clusters, but other parallel computing platforms are emerging. For example, the performance trade-offs for manycore SMP and non-cache-coherent processors are very different from a cluster, but these platforms can also benefit from a simpler programming model than shared-memory multi-threading or explicit message passing. Therefore we are developing a more lightweight version of CIEL that can support finer-grained parallelism on multicore machines. We are also applying the ideas of Skywriting to other languages, including Python, Java, Scala, and OCaml, with the aim of combining coordination and computation in a single, efficient language. Ultimately, we predict that future clusters will be built from servers containing manycore processors, and the techniques we have described here will be useful for dealing with parallelism at multiple scales.

Acknowledgments and References

We would like to thank the other members of the CIEL team: Anil Madhavapeddy, Malte Schwarzkopf, Steven Smith, and Chris Smowton.

A deeper introduction to CIEL and Skywriting, including implementation details and performance evaluation, can be found in Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand, "CIEL: A Universal Execution Engine for Distributed Data-Flow Computing," in *Proceedings of NSDI '11*.

CIEL and Skywriting are available for download from our project Web site, which includes tutorials and example applications: http://www.cl.cam.ac.uk/netos/ciel/.