CIEL: a universal execution engine for distributed data-flow computing

Derek G. Murray Malte Schwarzkopf Christopher Smowton
Steven Smith Anil Madhavapeddy Steven Hand
University of Cambridge Computer Laboratory

Abstract

This paper introduces CIEL, a universal execution engine for distributed data-flow programs. Like previous execution engines, CIEL masks the complexity of distributed programming. Unlike those systems, a CIEL job can make data-dependent control-flow decisions, which enables it to compute iterative and recursive algorithms.

We have also developed Skywriting, a Turing-complete scripting language that runs directly on CIEL. The execution engine provides transparent fault tolerance and distribution to Skywriting scripts and high-performance code written in other programming languages. We have deployed CIEL on a cloud computing platform, and demonstrate that it achieves scalable performance for both iterative and non-iterative algorithms.

1 Introduction

Many organisations have an increasing need to process large data sets, and a cluster of commodity machines on which to process them. *Distributed execution engines*—such as MapReduce [18] and Dryad [26]—have become popular systems for exploiting such clusters. These systems expose a simple programming model, and automatically handle the difficult aspects of distributed computing: fault tolerance, scheduling, synchronisation and communication. MapReduce and Dryad can be used to implement a wide range of algorithms [3, 39], but they are awkward or inefficient for others [12, 21, 25, 28, 34]. The problems typically arise with *iterative* algorithms, which underlie many machine-learning and optimisation problems, but require a more *expressive* programming model and a more powerful execution engine. To address these limitations, and extend the benefits of distributed execution engines to a wider range of applications, we have developed Skywriting and CIEL.

Skywriting is a scripting language that allows the straightforward expression of iterative and recursive task-parallel algorithms using imperative and functional language syntax [31]. Skywriting scripts run on CIEL, an execution engine that provides a *universal* execution model for distributed data-flow. Like previous systems, CIEL coordinates the distributed execution of a set of data-parallel tasks arranged according to a data-flow DAG, and hence benefits from transparent scaling and fault tolerance. However CIEL extends previous models by *dynamically* building the DAG as tasks execute. As we will show, this conceptually simple extension—allowing tasks to create further tasks—enables CIEL to support data-dependent iterative or recursive algorithms. We present the high-level architecture of CIEL in Section 3, and explain how Skywriting maps onto CIEL's primitives in Section 4.

Our implementation incorporates several additional features, described in Section 5. Like existing systems, CIEL provides transparent fault tolerance for worker nodes. Moreover, CIEL can tolerate failures of the cluster master and the client program. To improve resource utilisation and reduce execution latency, CIEL can memoise the results of tasks. Finally, CIEL supports the streaming of data between concurrently-executing tasks.

We have implemented a variety of applications in Skywriting, including MapReduce-style (grep, word-count), iterative (k-means, PageRank) and dynamic-programming (Smith-Waterman, option pricing) algorithms. In Section 6 we evaluate the performance of some of these applications when run on a CIEL cluster.

2 Motivation

Several researchers have identified limitations in the MapReduce and Dryad programming models. These systems were originally developed for batch-oriented jobs, namely large-scale text mining for information retrieval [18, 26]. They are designed to maximise throughput, rather than minimise individual job latency. This is especially noticeable in iterative computations, for which

Feature	MapReduce [2, 18]	Dryad [26]	Pregel [28]	Iterative MR [12, 21]	Piccolo [34]	CIEL
Dynamic control flow	✗	✗	✓	✓	✓	✓
Task dependencies	Fixed (2-stage)	Fixed (DAG)	Fixed (BSP)	Fixed (2-stage)	Fixed (1-stage)	Dynamic
Fault tolerance	Transparent	Transparent	Transparent	✗	Checkpoint	Transparent
Data locality	✓	✓	✓	✓	✓	✓
Transparent scaling	✓	✓	✓	✓	✗	✓

Figure 1: Analysis of the features provided by existing distributed execution engines.

multiple jobs are chained together and the job latency is multiplied [12, 21, 25, 28, 34].

Nevertheless, MapReduce—in particular its open-source implementation, Hadoop [2]—remains a popular platform for parallel iterative computations with large inputs. For example, the Apache Mahout machine learning library uses Hadoop as its execution engine [3]. Several of the Mahout algorithms—such as k-means clustering and singular value decomposition—are iterative, comprising a data-parallel kernel inside a while-not-converged loop. Mahout uses a *driver program* that submits multiple jobs to Hadoop and performs convergence testing at the client. However, since the driver program executes logically (and often physically) outside the Hadoop cluster, each iteration incurs job-submission overhead, and the driver program does not benefit from transparent fault tolerance. These problems are not unique to Hadoop, but are shared with both the original version of MapReduce [18] and Dryad [26].

The computational power of a distributed execution engine is determined by the data flow that it can express. In MapReduce, the data flow is limited to a bipartite graph parameterised by the number of map and reduce tasks; Dryad allows data flow to follow a more general directed acyclic graph (DAG), but it must be fully specified before starting the job. In general, to support iterative or recursive algorithms within a single job, we need *data-dependent control flow*—i.e. the ability to create more work dynamically, based on the results of previous computations. At the same time, we wish to retain the existing benefits of task-level parallelism: transparent fault tolerance, locality-based scheduling and transparent scaling. In Figure 1, we analyse a range of existing systems in terms of these objectives.

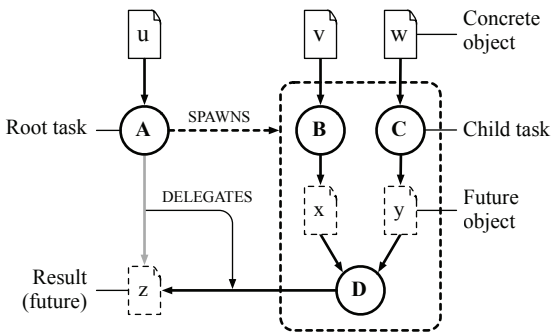
MapReduce and Dryad already support transparent fault tolerance, locality-based scheduling and transparent scaling [18, 26]. In addition, Dryad supports arbitrary task dependencies, which enables it to execute a larger class of computations than MapReduce. However, neither supports data-dependent control flow, so the work in each computation must be statically pre-determined.

A variety of systems provide data-dependent control flow but sacrifice other functionality. Google's Pregel is the largest-scale example of a distributed execution engine with support for control flow [28]. Pregel is a Bulk Synchronous Parallel (BSP) system designed for executing graph algorithms (such as PageRank), and Pregel computations are divided into "supersteps", during which a "vertex method" is executed for each vertex in the graph. Crucially, each vertex can vote to terminate the computation, and the computation terminates when all vertices vote to terminate. Like a simple MapReduce job, however, a Pregel computation only operates on a single data set, and the programming model does not support the composition of multiple computations.

Two recent systems add iteration capabilities to MapReduce. CGL-MapReduce is a new implementation of MapReduce that caches static (loop-invariant) data in RAM across several MapReduce jobs [21]. HaLoop extends Hadoop with the ability to evaluate a convergence function on reduce outputs [12]. Neither system provides fault tolerance across multiple iterations, and neither can support Dryad-style task dependency graphs.

Finally, Piccolo is a new programming model for data-parallel programming that uses a partitioned in-memory key-value table to replace the reduce phase of MapReduce [34]. A Piccolo program is divided into "kernel" functions, which are applied to table partitions in parallel, and typically write key-value pairs into one or more other tables. A "control" function coordinates the kernel functions, and it may perform arbitrary data-dependent control flow. Piccolo supports user-assisted checkpointing (based on the Chandy-Lamport algorithm), and is limited to fixed cluster membership. If a single machine fails, the entire computation must be restarted from a checkpoint with the same number of machines.

We believe that CIEL is the first system to support all five goals in Figure 1, but it is not a panacea. CIEL is designed for coarse-grained parallelism across large data sets, as are MapReduce and Dryad. For fine-grained tasks, a work-stealing scheme is more appropriate [11]. Where the entire data set can fit in RAM, Piccolo may be more efficient, because it can avoid writing to disk. Ultimately, achieving the highest performance requires significant developer effort, using a low-level technique such as explicit message passing [30].



(a) Dynamic task graph

Task ID	Dependencies	Expected outputs
A	{ u }	~~z~~
B	{ v }	x
C	{ w }	y
D	{ x, y }	z

Object ID	Produced by	Locations
u	–	{ `host19`, `host85` }
v	–	{ `host21`, `host23` }
w	–	{ `host22`, `host57` }
x	**B**	∅
y	**C**	∅
z	~~**A**~~ **D**	∅

(b) Task and object tables

Figure 2: A CIEL job is represented by a dynamic task graph, which contains tasks and objects (§3.1). In this example, root task **A** spawns tasks **B**, **C** and **D**, and delegates the production of its result to **D**. Internally, CIEL uses task and object tables to represent the graph (§3.3).

3 CIEL

CIEL is a distributed execution engine that can execute programs with arbitrary data-dependent control flow. In this section, we first describe the core abstraction that CIEL supports: the *dynamic task graph* (§3.1). We then describe how CIEL executes a job that is represented as a dynamic task graph (§3.2). Finally, we describe the concrete architecture of a CIEL cluster that is used for distributed data-flow computing (§3.3).

3.1 Dynamic task graphs

In this subsection, we define the three CIEL primitives—objects, references and tasks—and explain how they are related in a dynamic task graph (Figure 2).

CIEL is a data-centric execution engine: the goal of a CIEL job is to produce one or more output **objects**. An object is an unstructured, finite-length sequence of bytes. Every object has a unique *name*: if two objects exist with the same name, they must have the same contents. To simplify consistency and replication, an object is immutable once it has been written, but it is sometimes possible to append to an object (§5.3).

It is helpful to be able to describe an object without possessing its full contents; CIEL uses **references** for this purpose. A reference comprises a name and a set of locations (e.g. hostname-port pairs) where the object with that name is stored. The set of locations may be empty: in that case, the reference is a *future reference* to an object that has not yet been produced. Otherwise, it is a *concrete reference*, which may be consumed.

A CIEL job makes progress by executing **tasks**. A task is a non-blocking atomic computation that executes completely on a single machine. A task has one or more *dependencies*, which are represented by references, and the task becomes runnable when all of its dependencies become concrete. The dependencies include a special object that specifies the behaviour of the task (such as an executable binary or a Java class) and may impose some structure over the other dependencies. To simplify fault tolerance (§5.2), CIEL requires that all tasks compute a deterministic function of their dependencies. A task also has one or more *expected outputs*, which are the names of objects that the task will either create or delegate another task to create.

Tasks can have two externally-observable behaviours. First, a task can **publish** one or more objects, by creating a concrete reference for those objects. In particular, the task can publish objects for its expected outputs, which may cause other tasks to become runnable if they depend on those outputs. To support data-dependent control flow, however, a task may also **spawn** new tasks that perform additional computation. CIEL enforces the following conditions on task behaviour:

1. For each of its expected outputs, a task must either publish a concrete reference, or spawn a child task with that name as an expected output. This ensures that, as long as the children eventually terminate, any task that depends on the parent's output will eventually become runnable.

2. A child task must only depend on concrete references (i.e. objects that already exist) or future references to the outputs of tasks that have already been spawned (i.e. objects that are already expected to be published). This prevents deadlock, as a cycle cannot form in the dependency graph.

The **dynamic task graph** stores the relation between tasks and objects. An edge from an object to a task means

that the task depends on that object. An edge from a task to an object means that the task is expected to output the object. As a job runs, new tasks are added to the dynamic task graph, and the edges are rewritten when a newly-spawned task is expected to produce an object.

The dynamic task graph provides low-level data-dependent control flow that resembles tail recursion: a task either produces its output (analogous to returning a value) or spawns a new task to produce that output (analogous to a tail call). It also provides facilities for data-parallelism, since independent tasks can be dispatched in parallel. However, we do not expect programmers to construct dynamic task graphs manually, and instead we provide the Skywriting script language for generating these graphs programmatically (§4).

3.2 Evaluating objects

Given a dynamic task graph, the role of CIEL is to evaluate one or more objects that correspond to the job outputs. Indeed, a CIEL job can be specified as a single *root task* that has only concrete dependencies, and an expected output that names the final result of the computation. This leads to two natural strategies, which are variants of topological sorting:

Eager evaluation. Since the task dependencies form a DAG, at least one task must have only concrete dependencies. Start by executing the tasks with only concrete dependencies; subsequently execute tasks when all of their dependencies become concrete.

Lazy evaluation. Seek to evaluate the expected output of the root task. To evaluate an object, identify the task, T, that is expected to produce the object. If T has only concrete dependencies, execute it immediately; otherwise, block T and recursively evaluate all of its unfulfilled dependencies using the same procedure. When the inputs of a blocked task become concrete, execute it. When the production of a required object is delegated to a spawned task, re-evaluate that object.

When we first developed CIEL, we experimented with both strategies, but switched exclusively to lazy evaluation since it more naturally supports the fault-tolerance and memoisation features that we describe in §5.

3.3 System architecture

Figure 3 shows the architecture of a CIEL cluster. A single *master* coordinates the end-to-end execution of jobs, and several *workers* execute individual tasks.

The master maintains the current state of the dynamic task graph in the *object table* and *task table* (Figure 2(b)).

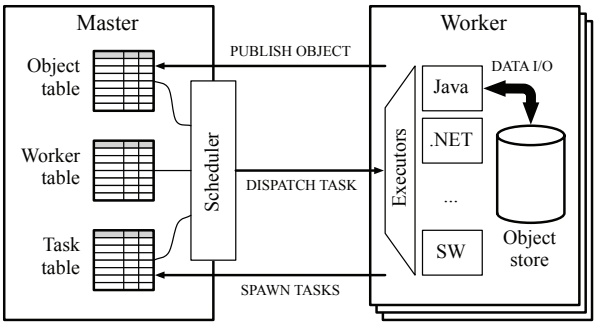


Figure 3: A CIEL cluster has a single master and many workers. The master dispatches tasks to the workers for execution. After a task completes, the worker publishes a set of objects and may spawn further tasks.

Each row in the object table contains the latest reference for that object, including its locations (if any), and a pointer to the task that is expected to produce it (if any: an object will not have a task pointer if it is loaded into the cluster by an external tool). Each row in the task table corresponds to a spawned task, and contains pointers to the references on which the task depends.

The master *scheduler* is responsible for making progress in a CIEL computation: it lazily evaluates output objects and pairs runnable tasks with idle workers. Since task inputs and outputs may be very large (on the order of gigabytes per task), all bulk data is stored on the workers themselves, and the master handles references. The master uses a multiple-queue-based scheduler (derived from Hadoop [2]) to dispatch tasks to the worker nearest the data. If a worker needs to fetch a remote object, it reads the object directly from another worker.

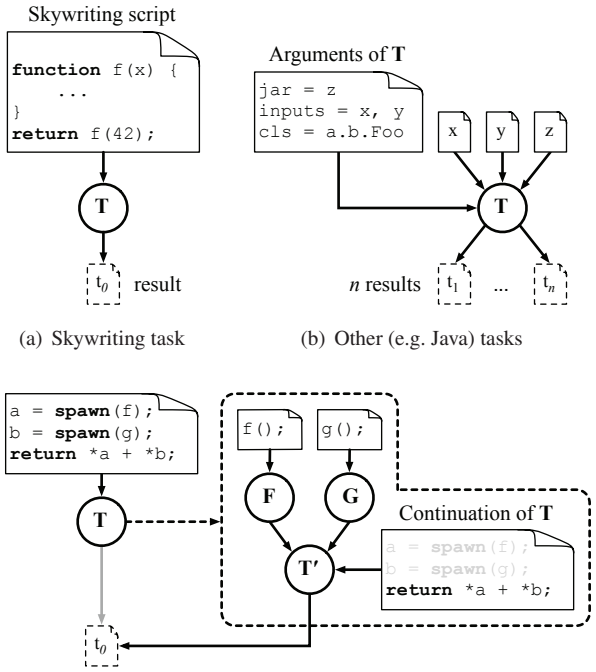
The workers execute tasks and store objects. At startup, a worker registers with the master, and periodically sends a heartbeat to demonstrate its continued availability. When a task is dispatched to a worker, the appropriate *executor* is invoked. An executor is a generic component that prepares input data for consumption and invokes some computation on it, typically by executing an external process. We have implemented simple executors for Java, .NET, shell-based and native code, as well as a more complex executor for Skywriting (§4).

Assuming that a worker executes a task successfully, it will reply to the master with the set of references that it wishes to publish, and a list of task descriptors for any new tasks that it wishes to spawn. The master will then update the object table and task table, and re-evaluate the set of tasks now runnable.

In addition to the master and workers, there will be one or more *clients* (not shown). A client's role is minimal: it submits a job to the master, and either polls the master to discover the job status or blocks until the job completes.

```
function process_chunk(chunk, prev_result) {
  // Execute native code for chunk processing.
  // Returns a reference to a partial result.
  return spawn_exec(...);
}

function is_converged(curr_result, prev_result) {
  // Execute native code for convergence test.
  // Returns a reference to a boolean.
  return spawn_exec(...)[0];
}

input_data = [ref("ciel://host137/chunk0"),
              ref("ciel://host223/chunk1"),
              ...];
curr = ...; // Initial guess at the result.

do {
  prev = curr;
  curr = [];
  for (chunk in input_data) {
    curr += process_chunk(chunk, prev);
  }
} while (!*is_converged(curr, prev));

return curr;
```

Figure 4: Iterative computation implemented in Skywriting. `input_data` is a list of n input chunks, and `curr` is initialised to a list of n partial results.



(a) Skywriting task

(b) Other (e.g. Java) tasks

(c) Implicit continuation due to dereferencing

Figure 5: Task creation in Skywriting. Tasks can be created using (a) `spawn()`, (b) `spawn_exec()` and (c) the dereference ($*$) operator.

A job submission message contains a root task, which must have only concrete dependencies. The master adds the root task to the task table, and starts the job by lazily evaluating its output (§3.2).

Note that CIEL currently uses a single (active) master for simplicity. Despite this, our implementation can recover from master failure (§5.2), and it did not cause a performance bottleneck during our evaluation (§6). Nonetheless, if it became a concern in future, it would be possible to partition the master state—i.e. the task table and object table—between several hosts, while retaining the functionality of a single logical master.

4 Skywriting

Skywriting is a language for expressing task-level parallelism that runs on top of CIEL. Skywriting is Turing-complete, and can express arbitrary data-dependent control flow using constructs such as `while` loops and recursive functions. Figure 4 shows an example Skywriting script that computes an iterative algorithm; we use a similar structure in the k-means experiment (§6.2).

We introduced Skywriting in a previous paper [31], but briefly restate the key features here:

- `ref(url)` returns a *reference* to the data stored at the given URL. The function supports common URL schemes, and the custom `ciel` scheme, which accesses entries in the CIEL object table. If the URL is external, CIEL downloads the data into the cluster as an object, and assigns a name for the object.

- `spawn(f, [arg, ...])` spawns a parallel task to evaluate `f(arg, ...)`. Skywriting functions are *pure*: functions cannot have side-effects, and all arguments are passed by value. The return value is a reference to the result of `f(arg, ...)`.

- `exec(executor, args, n)` synchronously runs the named `executor` with the given `args`. The executor will produce `n` outputs. The return value is a list of `n` references to those outputs.

- `spawn_exec(executor, args, n)` spawns a parallel task to run the named `executor` with the given `args`. As with `exec()`, the return value is a list of `n` references to those outputs.

- The dereference (unary-$*$) operator can be applied to any reference; it loads the referenced data into the Skywriting execution context, and evaluates to the resulting data structure.

In the following, we describe how Skywriting maps on to CIEL primitives. We describe how tasks are created (§4.1), how references are used to facilitate data-dependent control flow (§4.2), and the relationship between Skywriting and other frameworks (§4.3).

4.1 Creating tasks

The distinctive feature of Skywriting is its ability to spawn new tasks in the middle of executing a job. The language provides two explicit mechanisms for spawning new tasks (the `spawn()` and `spawn_exec()` functions) and one implicit mechanism (the $\star$-operator). Figure 5 summarises these mechanisms.

The `spawn()` function creates a new task to run the given Skywriting function. To do this, the Skywriting runtime first creates a data object that contains the new task's environment, including the text of the function to be executed and the values of any arguments passed to the function. This object is called a Skywriting *continuation*, because it encapsulates the state of a computation. The runtime then creates a task descriptor for the new task, which includes a dependency on the new continuation. Finally, it assigns a reference for the task result, which it returns to the calling script. Figure 5(a) shows the structure of the created task.

The `spawn_exec()` function is a lower-level task-creation mechanism that allows the caller to invoke code written in a different language. Typically, this function is not called directly, but rather through a wrapper for the relevant executor (e.g. the built-in `java()` library function). When `spawn_exec()` is called, the runtime serialises the arguments into a data object and creates a task that depends on that object (Figure 5(b)). If the arguments to `spawn_exec()` include references, the runtime adds those references to the new task's dependencies, to ensure that CIEL will not schedule the task until all of its arguments are available. Again, the runtime creates references for the task outputs, and returns them to the calling script. We discuss how names are chosen in §5.1.

If the task attempts to dereference an object that has not yet been created—for example, the result of a call to `spawn()`—the current task must block. However, CIEL tasks are *non-blocking*: all synchronisation (and data-flow) must be made explicit in the dynamic task graph (§3.1). To resolve this contradiction, the runtime implicitly creates a *continuation task* that depends on the dereferenced object and the current continuation (i.e. the current Skywriting execution stack). The new task therefore will only run when the dereferenced object has been produced, which provides the necessary synchronisation. Figure 5(c) shows the dependency graph that results when a task dereferences the result of `spawn()`.

A task terminates when it reaches a `return` statement (or it blocks on a future reference). A Skywriting task has a single output, which is the value of the expression in the `return` statement. On termination, the runtime stores the output in the local object store, publishes a concrete reference to the object, and sends a list of spawned tasks to the master, in order of creation.

Skywriting ensures that the dynamic task graph remains acyclic. A task's dependencies are fixed when the task-creation function is evaluated, which means that they can only include references that are stored in the local Skywriting scope before evaluating the function. Therefore, a task cannot depend on itself or any of its descendants. Note that the results of `spawn()` and `spawn_exec()` are first-class *futures* [24]: a Skywriting task can pass the references in its return value or in a subsequent call to the task-creation functions. This enables a script to create arbitrary acyclic dependency graphs, such as the MapReduce dependency graph (§4.3).

4.2 Data-dependent control flow

Skywriting is designed to coordinate *data-centric* computations, which means that the objects in the computation can be divided into two spaces:

Data space. Contains large data objects that may be up to several gigabytes in size.

Coordination space. Contains small objects—such as integers, booleans, strings, lists and dictionaries—that determine the control flow.

In general, objects in the data space are processed by programs written in compiled languages, to achieve better I/O or computational performance than Skywriting can provide. In existing distributed execution engines (such as MapReduce and Dryad), the data space and coordination space are disjoint, which prevents these systems from supporting data-dependent control flow.

To support data-dependent control flow, data must be able to pass from the data space into the coordination space, so that it can help to determine the control flow. In Skywriting, the $\star$-operator transforms a reference to a (data space) object into a (coordination space) value. The producing task, which may be run by any executor, must write the referenced object in a format that Skywriting can recognise; we use JavaScript Object Notation (JSON) for this purpose [4]. This serialisation format is only used for references that are passed to Skywriting, and the majority of executors use the appropriate binary format for their data.

4.3 Other languages and frameworks

Systems like MapReduce have become popular, at least in part, because of their simple interface: a developer can specify a whole distributed computation with just a pair of `map()` and `reduce()` functions. To demonstrate that Skywriting approaches this level of simplicity, Figure 6 shows an implementation of the MapReduce execution model, taken from the Skywriting standard library.

```
function apply(f, list) {
  outputs = [];
  for (i in range(len(list))) {
    outputs[i] = f(list[i]);
  }
  return outputs;
}

function shuffle(inputs, num_outputs) {
  outputs = [];
  for (i in range(num_outputs)) {
    outputs[i] = [];
    for (j in range(len(inputs))) {
      outputs[i][j] = inputs[j][i];
    }
  }
  return outputs;
}

function mapreduce(inputs, mapper, reducer, r) {
  map_outputs = apply(mapper, inputs);
  reduce_inputs = shuffle(map_outputs, r);
  reduce_outputs = apply(reducer, reduce_inputs);
  return reduce_outputs;
}
```

Figure 6: Implementation of the MapReduce programming model in Skywriting. The user provides a list of inputs, a mapper function, a reducer function and the number of reducers to use.

The `mapreduce()` function first applies `mapper` to each element of `inputs`. `mapper` is a Skywriting function that returns a list of `r` elements. The map outputs are then shuffled, so that the i^{th} output of each map becomes an input to the i^{th} reduce. Finally, the `reducer` function is applied `r` times to the collected reduce inputs. In typical use, the inputs to `mapreduce()` are data objects containing the input splits, and the `mapper` and `reducer` functions invoke `spawn_exec()` to perform computation in another language.

Note that the `mapper` function is responsible for partitioning data amongst the reducers, and the `reducer` function must merge the inputs that it receives. The implementation of `mapper` may also incorporate a combiner, if desired [18]. To simplify development, we have ported portions of the Hadoop MapReduce framework to run as CIEL tasks, and provide helper functions for partitioning, merging, and processing Hadoop file formats.

Any higher-level language that is compiled into a DAG of tasks can also be compiled into a Skywriting program, and executed on a CIEL cluster. For example, one could develop Skywriting back-ends for Pig [32] and DryadLINQ [39], raising the possibility of extending those languages with support for unbounded iteration.

5 Implementation issues

The current implementation of CIEL and Skywriting contains approximately 9,500 lines of Python code, and a few hundred lines of C, Java and other languages in the executor bindings. All of the source code, along with a suite of example Skywriting programs (including those used to evaluate the system in §6), is available to download from our project website:

`http://www.cl.cam.ac.uk/netos/ciel/`

The remainder of this section describes three interesting features of our implementation: memoisation (§5.1), master fault tolerance (§5.2) and streaming (§5.3).

5.1 Deterministic naming & memoisation

Recall that all objects in a CIEL cluster have a *unique* name. In this subsection, we show how an appropriate choice of names can enable memoisation.

Our original implementation of CIEL used globally-unique identifiers (UUIDs) to identify all data objects. While this was a conceptually simple scheme, it complicated fault tolerance (see following subsection), because the master had to record the generated UUIDs to support deterministic task replay after a failure.

This motivated us to reconsider the choice of names. To support fault-tolerance, existing systems assume that individual tasks are deterministic [18, 26], and CIEL makes the same assumption (§3.1). It follows that two tasks with the same dependencies—including the executable code as a dependency—will have identical behaviour. Therefore the n outputs of a task created with the following Skywriting statement

```
result = spawn_exec(executor, args, n);
```

will be completely determined by `executor`, `args`, `n` and their indices. We could therefore construct a name for the i^{th} output by concatenating `executor`, `args`, `n` and i, with appropriate delimiters. However, since `args` may itself contain references, names could grow to an unmanageable length. We therefore use a collision-resistant hash function, $\mathcal{H}$, to compute a digest of `args` and `n`, which gives the resulting name:

executor	:	$\mathcal{H}$(args||n)	:	i

We currently use the 160-bit SHA-1 hash function to generate the digest.

Recall the lazy evaluation algorithm from §3.2: tasks are only executed when their expected outputs are needed to resolve a dependency for a blocked task. If a new task's outputs have already been produced by a previous task, the new task need not be executed at all. Hence, as a result of deterministic naming, CIEL memoises task results, which can improve the performance of jobs that perform repetitive tasks.

The goals of our memoisation scheme are similar to the recent Nectar system [23]. Nectar performs static

analysis on DryadLINQ queries to identify subqueries that have previously been computed on the same data. Nectar is implemented at the DryadLINQ level, which enables it to make assumptions about the semantics of the each task, and the cost/benefit ratio of caching intermediate results. For example, Nectar can re-use the results of commutative and associative aggregations from a previous query, if the previous query operated on a prefix of the current query's input. The expressiveness of CIEL jobs makes it more challenging to run these analyses, and we are investigating how simple annotations in a Skywriting program could provide similar functionality in our system.

5.2 Fault tolerance

A distributed execution engine must continue to make progress in the face of network and computer faults. As jobs become longer—and, since CIEL allows unbounded iteration, they may become extremely long—the probability of experiencing a fault increases. Therefore, CIEL must tolerate the failure of any machine involved in the computation: the client, workers and master.

Client fault tolerance is trivial, since CIEL natively supports iterative jobs and manages job execution from start to finish. The client's only role is to submit the job: if the client subsequently fails, the job will continue without interruption. By contrast, in order to execute an iterative job using a non-iterative framework, the client must run a driver program that performs all data-dependent control flow (such as convergence testing). Since the driver program executes outside the framework, it does not benefit from transparent fault tolerance, and the developer must provide this manually, for example by checkpointing the execution state. In our system, a Skywriting script replaces the driver program, and CIEL executes the whole script reliably.

Worker fault tolerance in CIEL is similar to Dryad [26]. The master receives periodic heartbeat messages from each worker, and considers a worker to have failed if (*i*) it has not sent a heartbeat after a specified timeout, and (*ii*) it does not respond to a reverse message from the master. At this point, if the worker has been assigned a task, that task is deemed to have failed.

When a task fails, CIEL automatically re-executes it. However, if it has failed because its inputs were stored on a failed worker, the task is no longer runnable. In that case, CIEL recursively re-executes predecessor tasks until all of the failed task's dependencies are resolved. To achieve this, the master invalidates the locations in the object table for each missing input, and lazily re-evaluates the missing inputs. Other tasks that depend on data from the failed worker will also fail, and these are similarly re-executed by the master.

Master fault tolerance is also supported in CIEL. In MapReduce and Dryad, a job fails completely if its master process fails [18, 26]; in Hadoop, all jobs fail if the JobTracker fails [2]; and master failure will usually cause driver programs that submit multiple jobs to fail. However, in CIEL, all master state can be derived from the set of active jobs. At a minimum, persistently storing the root task of each active job allows a new master to be created and resume execution immediately. CIEL provides three complementary mechanisms that extend master fault tolerance: *persistent logging*, *secondary masters* and *object table reconstruction*.

When a new job is created, the master creates a log file for the job, and synchronously writes its root task descriptor to the log. By default, it writes the log to a log directory on local secondary storage, but it can also write to a networked file system or distributed storage service. As new tasks are created, their descriptors are appended asynchronously to the log file, and periodically flushed to disk. When the job completes, a concrete reference to its result is written to the log directory. Upon restarting, the master scans its log directory for jobs without a matching result. For those jobs, it replays the log, rebuilding the dynamic task graph, and ignoring the final record if it is truncated. Once all logs have been processed, the master restarts the jobs by lazily evaluating their outputs.

Alternatively, the master may log state updates to a *secondary master*. After the secondary master registers with the primary master, the primary asynchronously forwards all task table and object table updates to the secondary. Each new job is sent synchronously, to ensure that it is logged at the secondary before the client receives an acknowledgement. In addition, the secondary records the address of every worker that registers with the primary, so that it can contact the workers in a fail-over scenario. The secondary periodically sends a heartbeat to the primary; when it detects that the primary has failed, the secondary instructs all workers to re-register with it. We evaluate this scenario in §6.5.

If the master fails and subsequently restarts, the workers can help to reconstruct the object table using the contents of their local object stores. A worker deems the master to have failed if it does not respond to requests. At this point, the worker switches into *reregister* mode, and the heartbeat messages are replaced with periodic registration requests to the same network location. When the worker finally contacts a new master, the master pulls a list of the worker's data objects, using a protocol based on GFS master recovery [22].

5.3 Streaming

Our earlier definition of a task (§3.1) stated that a task produces data objects as part of its *result*. This definition

implies that object production is atomic: an object either exists completely or not at all. However, since data objects may be very large, there is often the opportunity to *stream* the partially-written object between tasks, which can lead to pipelined parallelism.

If the producing task has streamable outputs, it sends a *pre-publish* message to the master, containing *stream references* for each streamable output. These references are used to update the object table, and may unblock other tasks: the *stream consumers*. A stream consumer executes as before, but the executed code reads its input from a named pipe rather than a local file. A separate thread in the consuming worker process fetches chunks of input from the producing worker, and writes them into the pipe. When the producer terminates successfully, it commits its outputs, which signals to the consumer that no more data remains to be read.

In the present implementation, the stream producer also writes its output data to a local disk, so that, if the stream consumer fails, the producer is unaffected. If the producer fails while it has a consumer, the producer rolls back any partially-written output. In this case, the consumer will fail due to missing input, and trigger re-execution of the producer (§5.2). We are investigating more sophisticated fault-tolerance and scheduling policies that would allow the producer and consumer to communicate via direct TCP streams, as in Dryad [26] and the Hadoop Online Prototype [16]. However, as we show in the following section, support for streaming yields useful performance benefits for some applications.

6 Evaluation

Our main goal in developing CIEL was to develop a system that supports a more powerful model of computation than existing distributed execution engines, without incurring a high cost in terms of performance. In this section, we evaluate the performance of CIEL running a variety of applications implemented in Skywriting. We investigate the following questions:

1. How does CIEL's performance compare to a system in production use (viz. Hadoop)? (§6.1, §6.2)
2. What benefits does CIEL provide when executing an iterative algorithm? (§6.2)
3. What overheads does CIEL impose on compute-intensive tasks? (§6.3, §6.4)
4. What effect does master failure have on end-to-end job performance? (§6.5)

For our evaluation, we selected a set of algorithms to answer these questions, including MapReduce-style, iterative, and compute-intensive algorithms. We chose dynamic programming algorithms to demonstrate CIEL's ability to execute algorithms with data dependencies that do not translate to the MapReduce model.

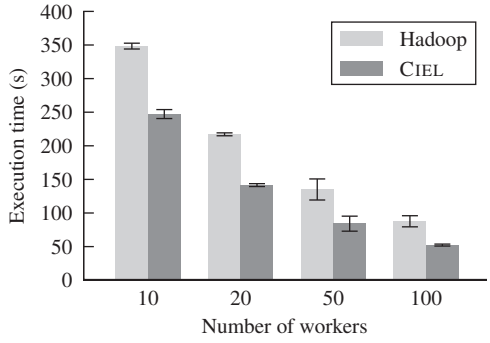


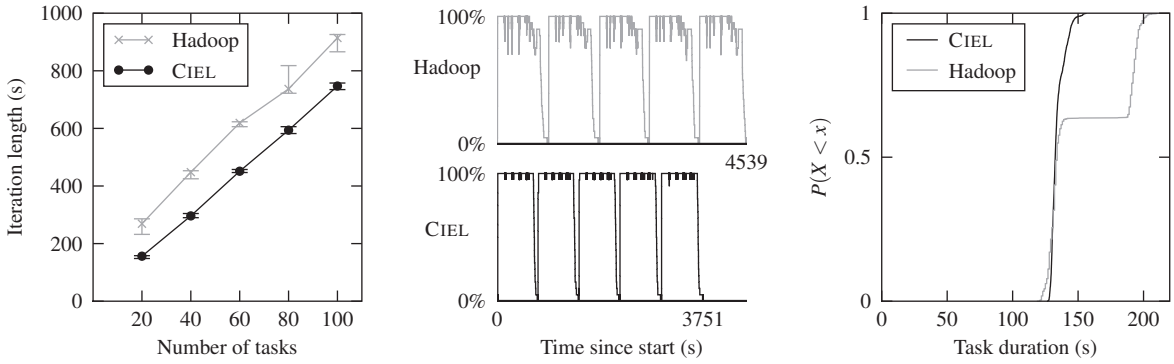
Figure 7: Grep execution time on Hadoop and CIEL (§6.1).

All of the results presented in this section were gathered using `m1.small` virtual machines on the Amazon EC2 cloud computing platform. At the time of writing, an `m1.small` instance has 1.7 GB of RAM and 1 virtual core (equivalent to a 2007 AMD Opteron or Intel Xeon processor) [1]. In all cases, the operating system was Ubuntu 10.04, using Linux kernel version 2.6.32 in 32-bit mode. Since the virtual machines are single-core, we run one CIEL worker per machine, and configure Hadoop to use one map slot per TaskTracker.

6.1 Grep

Our grep benchmark uses the Grep example application from Hadoop to search a 22.1 GB dump of English-language Wikipedia for a three-character string. The original Grep application performs two MapReduce jobs: the first job parses the input data and emits the matching strings, and the second sorts the matching strings by frequency. In Skywriting, we implemented this as a single script that uses two invocations of `mapreduce()` (§4.3). Both systems use identical data formats and execute an identical computation (regular expression matching).

Figure 7 shows the absolute execution time for Grep as the number of workers increases from 10 to 100. Averaged across all runs, CIEL outperforms Hadoop by 35%. We attribute this to the Hadoop heartbeat protocol, which limits the rate at which TaskTrackers poll for tasks once every 5 seconds, and the mandatory "setup" and "cleanup" phases that run at the start and end of each job [38]. As a result, the relative performance of CIEL improves as the job becomes shorter: CIEL takes 29% less time on 10 workers, and 40% less time on 100



(a) Iteration length

(b) Cluster utilisation (100 tasks)

(c) Map task distribution

Figure 8: Results of the k-means experiment on Hadoop and CIEL with 20 workers (§6.2).

workers. We observed that a no-op Hadoop job (which dispatches one map task per worker, and terminates immediately) runs for an average of 30 seconds. Since Grep involves two jobs, we would not expect Hadoop to complete the benchmark in less than 60 seconds. These results confirm that Hadoop is not well-suited to short jobs, which is a result of its original application (large-scale document indexing). However, anecdotal evidence suggests that production Hadoop clusters mostly run jobs lasting less than 90 seconds [40].

6.2 k-means

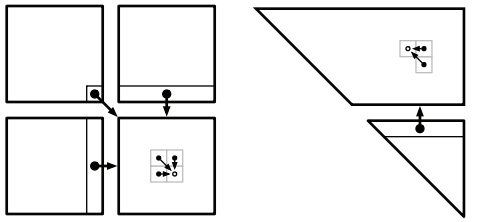
We ported the Hadoop-based k-means implementation from the Apache Mahout scalable machine learning toolkit [3] to CIEL. Mahout simulates iterative-algorithm support on Hadoop by submitting a series of jobs and performing a convergence test outside the cluster; our port uses a Skywriting script that performs all iterations and convergence testing in a single CIEL job.

In this experiment, we compare the performance of the two versions by running 5 iterations of clustering on 20 workers. Each task takes 64 MB of input—80,000 dense vectors, each containing 100 double-precision values—and $k = 100$ cluster centres. We increase the number of tasks from 20 to 100, in multiples of the cluster size. As before, both systems use identical data formats and execute an identical computational kernel. Figure 8(a) compares the per-iteration execution time for the two versions. For each job size, CIEL is faster than Hadoop, and the difference ranges between 113 and 168 seconds. To investigate this difference further, we now analyse the task execution profile.

Figure 8(b) shows the cluster utilisation as a function of time for the 5 iterations of 100 tasks. From this figure, we can compute the average cluster utilisation: i.e. the probability that a worker is assigned a task at any point during the job execution. Across all job sizes, CIEL achieves 89 $\pm$ 2% average utilisation, whereas Hadoop achieves 84% utilisation for 100 tasks (and only 59% utilisation for 20 tasks). The Hadoop utilisation drops to 70% at several points when there is still runnable work, which is visible as troughs or "noise" in the utilisation time series. This scheduling delay is due to Hadoop's polling-based implementation of task dispatch.

CIEL also achieves higher utilisation in this experiment because the task duration is less variable. The execution time of k-means is dominated by the map phase, which computes k Euclidean distances for each data point. Figure 8(c) shows the cumulative distribution of map task durations, across all k-means experiments. The Hadoop distribution is clearly bimodal, with 64% of the tasks being "fast" ($\mu = 130.9$, $\sigma = 3.92$) and 36% of the tasks being "slow" ($\mu = 193.5$, $\sigma = 3.71$). By contrast, all of the CIEL tasks are "fast" ($\mu = 134.1$, $\sigma = 5.05$). On closer inspection, the slow Hadoop tasks are non-data-local: i.e. they read their input from another HDFS data node. When computing an iterative job such as k-means, CIEL can use information about previous iterations to improve the performance of subsequent iterations. For example, CIEL preferentially schedules tasks on workers that consumed the same inputs in previous iterations, in order to exploit data that might still be stored in the page cache. When a task reads its input from a remote worker, CIEL also updates the object table to record that another replica of that input now exists. By contrast, each iteration on Hadoop is an independent job, and Hadoop does not perform cross-job optimisations, so the scheduler is less able to exploit data locality.

In the CIEL version, a Skywriting task performs a convergence test and, if necessary, spawns a subsequent iteration of k-means. However, compared to the data-intensive map phase, its execution time is insignificant: in the 100-task experiment, less than 2% of the total job



(a) Smith-Waterman (b) Binomial options pricing

Figure 9: Smith-Waterman (§6.3) and BOPM (§6.4) are dynamic programming algorithms, with macro-level (partition) and micro-level (element) dependencies.

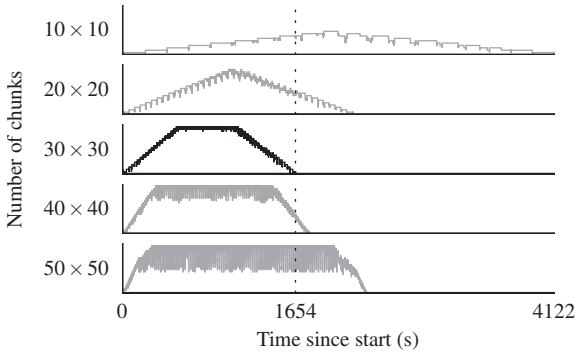


Figure 10: Smith-Waterman cluster utilisation against time, for different block granularities. The best performance is observed with 30 × 30 blocks.

execution time is spent running Skywriting tasks. The Skywriting execution time is dominated by communication with the master, as the script sends a new task descriptor to the master for each task in the new iteration.

6.3 Smith-Waterman

In this experiment, we evaluate strategies for parallelising the Smith-Waterman sequence alignment algorithm [36]. For strings of size m and n, the algorithm computes mn elements of a dynamic programming matrix. However, since each element depends on three predecessors, the algorithm is not embarrassingly parallel. We divide the matrix into blocks—where each block depends on values from its three neighbours (Figure 9(a))—and process one block per task.

We use CIEL to compute the alignment between two 1 MB strings on 20 workers. Figure 10 shows the cluster utilisation as the block granularity is varied: a granularity of $m \times n$ means that the computation is split into mn blocks. For 10 × 10 (the most coarse-grained case that we consider), the maximum degree of parallelism is 10, because the dependency structure limits the

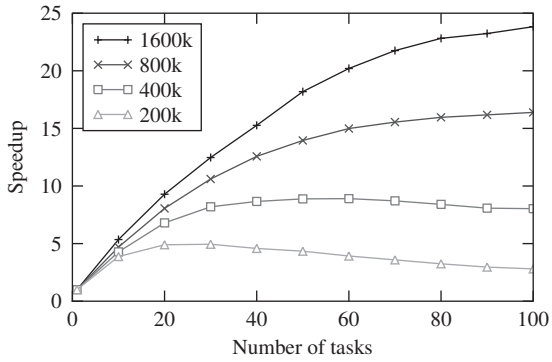


Figure 11: Speedup of BOPM (§6.4) on 47 workers as the number of tasks is varied and the resolution is increased.

maximum achievable parallelism to the length of the antidiagonal in the block matrix. Increasing the number of blocks to 20 × 20 allows CIEL to achieve full utilisation briefly, but performance remains poor because the majority of the job duration is spent either ramping up to or down from full utilisation. We observe the best performance for 30 × 30, which ramps up to full utilisation more quickly than coarser-grained configurations, and maintains full utilisation for an extended period, because there are more runnable tasks than workers. Increasing the granularity beyond 30 × 30 leads to poorer overall performance, because the overhead of task dispatch becomes a significant fraction of task duration. Furthermore, the scheduler cannot dispatch tasks quickly enough to maintain full utilisation, which appears as "noise" in Figure 10.

6.4 Binomial options pricing

We now consider another dynamic programming algorithm: the binomial options pricing model (BOPM) [17]. BOPM computes a binomial tree, which can be represented as an upper-triangular matrix, P. The rightmost column of P can be computed directly from the input parameters, after which element $p_{i,j}$ depends on $p_{i,j+1}$ and $p_{i+1,j+1}$, and the result is the value of $p_{1,1}$. We achieve parallelism by dividing the matrix into row chunks, creating one task per chunk, and *streaming* the top row of each chunk into the next task. Figure 9(b) shows the element- and chunk-level data dependencies for this algorithm.

BOPM is not an embarrassingly parallel algorithm. However, we expect CIEL to achieve some speedup, since rows of the matrix can be computed in parallel, and we can use streaming tasks (§5.3) to obtain pipelined parallelism. We can also achieve better speedup by increasing the resolution of the calculation: the problem size

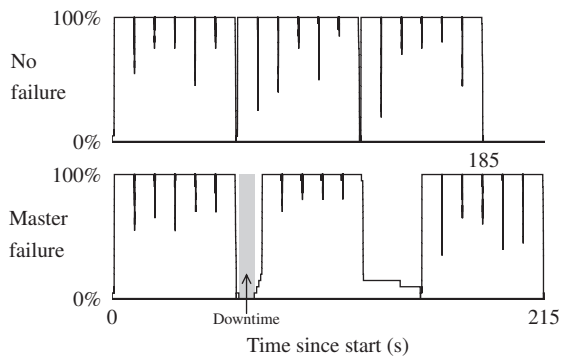


Figure 12: Cluster utilisation for three iterations of an iterative algorithm (§6.5). In the lower case, the primary master fails over to a secondary at the beginning of the second iteration. The total downtime is 7.7 seconds.

(n) is inversely proportional to the time step (Δt), and the serial execution time increases as $O(n^2)$.

Figure 11 shows the parallel speedup of BOPM on a 47-worker CIEL cluster. We vary the number of tasks, and increase n from 2×10^5 to 1.6×10^6. As expected, the maximum speedup increases as the problem size grows, because the amount of independent work in each task grows. For $n = 2 \times 10^5$ the maximum speedup observed is 4.9×, whereas for $n = 1.6 \times 10^6$ the maximum speedup observed is 23.8×. After reaching the maximum, the speedup decreases as more tasks are added, because small tasks suffer proportionately more from constant per-task overhead. Due to our streaming implementation, the minimum execution time for a stream consumer is approximately one second. We plan to replace our simple, polling-based streaming implementation with direct TCP sockets, which will decrease the per-task overhead and improve the maximum speedup.

6.5 Fault tolerance

Finally, we conducted an experiment in which master fail-over was induced during an iterative computation. Figure 12 contrasts the cluster utilisation in the non-failure and master-failure cases, where the master fail-over occurs at the beginning of the second iteration. Between the failure of the primary master and the resumption of execution, 7.7 seconds elapse: during this time, the secondary master must detect primary failure, contact all of the workers, and wait until the workers register with the secondary. Utilisation during the second iteration is poorer, because some tasks must be replayed due to the failure. The overall job execution time increases by 30 seconds, and the original full utilisation is attained once more in the third iteration.

7 Alternative approaches

CIEL was inspired primarily by the MapReduce and Dryad distributed execution engines. However, there are several different and complementary approaches to large-scale distributed computing. In this section, we briefly survey the related work from different fields.

7.1 High performance computing (HPC)

The HPC community has long experience in developing parallel programs. OpenMP is an API for developing parallel programs on shared-memory machines, which has recently added support for task parallelism with dependencies [7]. In this model, a task is a C or Fortran function marked with a compiler directive that identifies the formal parameters as task inputs and outputs. The inputs and outputs are typically large arrays that fit completely in shared memory. OpenMP is more suitable than CIEL for jobs that share large amounts of data that is frequently updated on a fine-grained basis. However, the parallel efficiency of a shared memory system is limited by interconnect contention and/or non-uniform memory access, which limits the practical size of an OpenMP job. Nevertheless, we could potentially use OpenMP to exploit parallelism within an individual multi-core worker.

Larger HPC programs typically use the Message Passing Interface (MPI) for parallel computing on distributed memory machines. MPI provides low-level primitives for sending and receiving messages, collective communication and synchronisation [30]. MPI is optimised for low-latency supercomputer interconnects, which often have a three-dimensional torus topology [35]. These interconnects are optimal for problems that decompose spatially and have local interactions with neighbouring processors. Since these interconnects are highly reliable, MPI does not tolerate intermittent message loss, and so checkpointing is usually used for fault tolerance. For example, Piccolo, which uses MPI, must restart an entire computation from a checkpoint if an error occurs [34].

7.2 Programming languages

Various programming paradigms have been proposed to simplify or fully automate software parallelisation.

Several projects have added parallel language constructs to existing programming languages. Cilk-NOW is a distributed version of Cilk that allows developers to `spawn` a C function on another cluster machine and `sync` on its result [11]. X10 is influenced by Java, and provides `finish` and `async` blocks that allow developers to implement more general synchronisation patterns [15]. Both implement *strict multithreading*, which restricts synchronisation to between a spawned thread

and its ancestor [10]. While this does not limit the expressiveness of these languages, it necessitates additional synchronisation in the implementation of, for example, MapReduce, where non-ancestor tasks may synchronise.

Functional programming languages offer the prospect of fully automatic parallelism [8]. NESL contains a parallel "apply to each" operator (i.e. a `map()` function) that processes the elements of a sequence in parallel, and the implementation allows nested invocation of this operator [9]. Glasgow Distributed Haskell contains mechanisms for remotely evaluating an expression on a particular host [33]. Though theoretically appealing, parallel functional languages have not demonstrated as great scalability as MapReduce or Dryad, which sacrifice expressivity for efficiency.

7.3 Declarative programming

The relational algebra, which comprises a relatively small set of operators, can be parallelised in time (pipelining) and space (partitioning) [19]. Pig and Hive implement the relational algebra using a DAG of MapReduce jobs on Hadoop [32, 37]; DryadLINQ and SCOPE implement it using a Dryad graph [14, 39].

The relational algebra is not universal but can be made more expressive by adding a least fixed point operator [5], and this research culminated in support for recursive queries in SQL:1999 [20]. Recently, Bu *et al.* showed how some recursive SQL queries may be translated to iterative Hadoop jobs [12].

Datalog is a declarative query language based on first-order logic [13]. Recently, Alvaro *et al.* developed a version of Hadoop and the Hadoop Distributed File System using Overlog (a dialect of Datalog), and demonstrated that it was almost as efficient as the equivalent Java code, while using far fewer lines of code [6]. We are not aware of any project that has used a fully-recursive logic-programming language to implement data-intensive programs, though the non-recursive Cascalog language, which runs on Hadoop, is a step in this direction [29].

7.4 Distributed operating systems

Hindman *et al.* have developed the Mesos distributed operating system to support "diverse cluster computing frameworks" on a shared cluster [25]. Mesos performs fine-grained scheduling and fair sharing of cluster resources between the frameworks. It is predicated on the idea that no single framework is suitable for all applications, and hence the resources must be virtualised to support different frameworks at once. By contrast, we have designed CIEL with primitives that support any form of computation (though not always optimally), and allow frameworks to be virtualised at the language level.

8 Conclusions

We designed CIEL to provide a superset of the features that existing distributed execution engines provide. With Skywriting, it it possible to write iterative algorithms in an imperative style and execute them with transparent fault tolerance and automatic distribution. However, CIEL can also execute any MapReduce job or Dryad graph, and the support for iteration allows it to perform Pregel- and Piccolo-style computations.

Our next step is to integrate CIEL primitives with existing programming languages. At present, only Skywriting scripts can create new tasks. This does not limit universality, but it requires developers to rewrite their driver programs in Skywriting. It can also put pressure on the Skywriting runtime, because all scheduling-related control-flow decisions must ultimately pass through interpreted code. The main benefit of Skywriting is that it masks the complexity of continuation-passing style behind the dereference operator (§4.2). We now seek a way to extend this abstraction to mainstream programming languages.

CIEL scales across hundreds of commodity machines, but other scaling challenges remain. For example, it is unclear how best to exploit multiple cores in a single machine, and we currently pass this problem to the executors, which receive full use of an individual machine. This gives application developers fine control over how their programs execute, at the cost of additional complexity. However, it limits efficiency if tasks are inherently sequential and multiple cores are available. Furthermore, the I/O saving from colocating a stream producer and its consumers on a single host may outweigh the cost of CPU contention. Finding the optimal schedule is a hard problem, and we are investigating simple annotation schemes and heuristics that improve performance in the common case. The recent work on cluster operating systems and scheduling algorithms [25, 27] offers hope that this problem will admit an elegant solution.

Further information about CIEL and Skywriting, including the source code, a language reference and a tutorial, is available from the project website:
`http://www.cl.cam.ac.uk/netos/ciel/`

Acknowledgements

We wish to thank our past and present colleagues in the Systems Research Group at the University of Cambridge for many fruitful discussions that contributed to the evolution of CIEL. We would also like to thank Byung-Gon Chun, our shepherd, and the anonymous reviewers, whose comments and suggestions have been invaluable for improving the presentation of this work.

References

[1] Amazon EC2. http://aws.amazon.com/ec2/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache Mahout. http://mahout.apache.org/.

[4] JSON. http://www.json.org/.

[5] AHO, A. V., AND ULLMAN, J. D. Universality of data retrieval languages. In *Proceedings of POPL* (1979).

[6] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. BOOM Analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of EuroSys* (2010).

[7] AYGUADÉ, E., COPTY, N., DURAN, A., HOEFLINGER, J., LIN, Y., MASSAIOLI, F., TERUEL, X., UNNIKRISHNAN, P., AND ZHANG, G. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst. 20*, 3 (2009), 404–418.

[8] BACKUS, J. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM 21*, 8 (1978), 613–641.

[9] BLELLOCH, G. E. Programming parallel algorithms. *Commun. ACM 39*, 3 (1996), 85–97.

[10] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

[11] BLUMOFE, R. D., AND LISIECKI, P. A. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of USENIX ATC* (1997).

[12] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. HaLoop: Efficient iterative data processing on large clusters. In *Proceedings of VLDB* (2010).

[13] CERI, S., GOTTLOB, G., AND TANCA, L. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng. 1*, 1 (1989), 146–166.

[14] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB* (2008).

[15] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA* (2005).

[16] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce Online. In *Proceedings of NSDI* (2010).

[17] COX, J. C., ROSS, S. A., AND RUBINSTEIN, M. Option pricing: A simplified approach. *Journal of Financial Economics 7*, 3 (1979), 229–263.

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI* (2004).

[19] DEWITT, D., AND GRAY, J. Parallel database systems: The future of high performance database systems. *Commun. ACM 35*, 6 (1992), 85–98.

[20] EISENBERG, A., AND MELTON, J. SQL: 1999, formerly known as SQL3. *SIGMOD Rec. 28*, 1 (1999), 131–138.

[21] EKANAYAKE, J., PALLICKARA, S., AND FOX, G. MapReduce for data intensive scientific analyses. In *Proceedings of eScience* (2008).

[22] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of SOSP* (2003).

[23] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in data centers. In *Proceedings of OSDI* (2010).

[24] HALSTEAD, JR., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[25] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI* (2011).

[26] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys* (2007).

[27] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of SOSP* (2009).

[28] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of SIGMOD* (2010).

[29] MARZ, N. Introducing Cascalog. http://nathanmarz.com/blog/introducing-cascalog/.

[30] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard. Tech. Rep. CS-94-230, University of Tennessee, 1994.

[31] MURRAY, D. G., AND HAND, S. Scripting the cloud with Skywriting. In *Proceedings of HotCloud* (2010).

[32] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD* (2008).

[33] POINTON, R. F., TRINDER, P. W., AND LOIDL, H.-W. The design and implementation of Glasgow Distributed Haskell. In *Proceedings of IFL* (2001).

[34] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of OSDI* (2010).

[35] SCOTT, S. L., AND THORSON, G. M. The Cray T3E network: adaptive routing in a high performance 3D torus. In *Proceedings of HOT Interconnects* (1996).

[36] SMITH, T., AND WATERMAN, M. Identification of common molecular subsequences. *Journal of molecular biology 147*, 1 (1981), 195–197.

[37] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTONY, S., LIU, H., AND MURTHY, R. Hive: A petabyte scale data warehouse using Hadoop. In *Proceedings of ICDE* (2010).

[38] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[39] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ÚLFAR ERLINGSSON, GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI* (2008).

[40] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys* (2010).