Reviewing:

CIEL

A universal execution engine for distributed data-flow computing

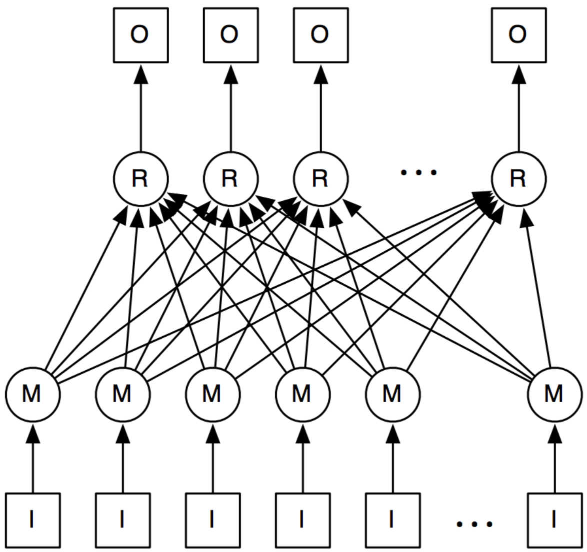
Presented by Niko Stahl for R202

Outline

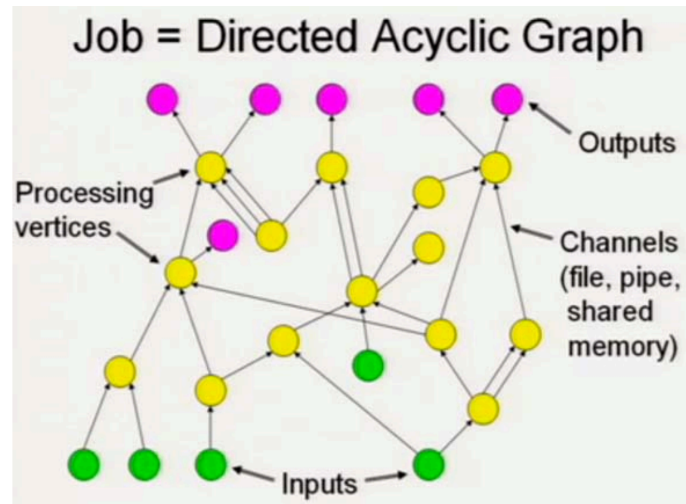
1. Motivation
2. Goals
3. Design
4. Fault Tolerance
5. Performance
6. Related Work
7. Conclusion

Motivation

MapReduce



Dryad



Motivation

MapReduce/Dryad have shortcomings:

1. Designed to maximize throughput, not to minimize latency.
2. Perform scheduling before running the algorithm. The resulting schedule is static.

These makes MapReduce/Dryad **inappropriate for iterative algorithms**.

Goals

Design a distributed execution framework that can

1. efficiently run iterative algorithms
2. provide a simple interface
3. offer transparent fault tolerance

Outline

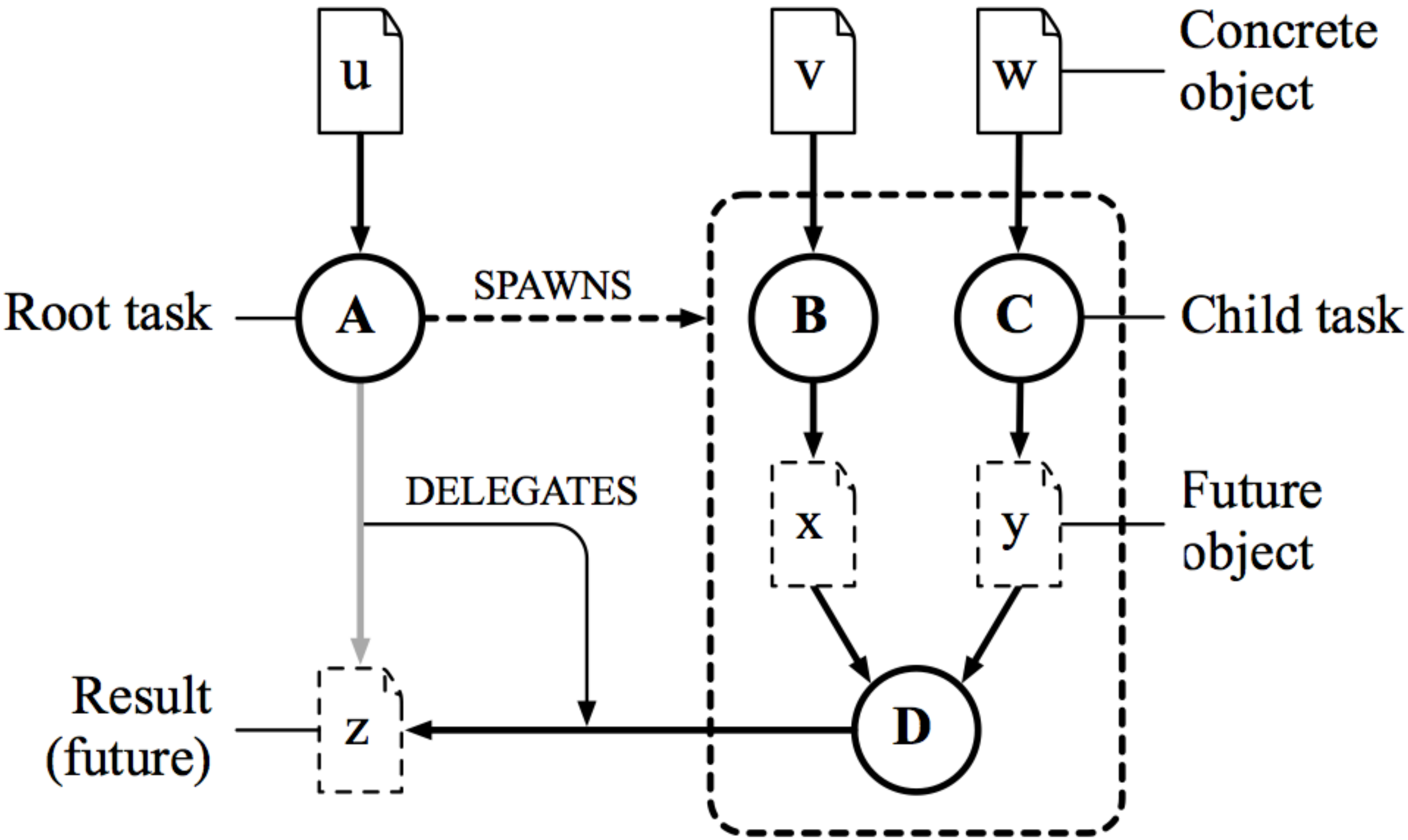
1. Motivation
2. Goals
3. Design
4. Fault Tolerance
5. Performance
6. Related Work
7. Conclusion

CIEL's Computation Model

The key feature of CIEL is a **dynamic task graph**.

Primitives of the model:

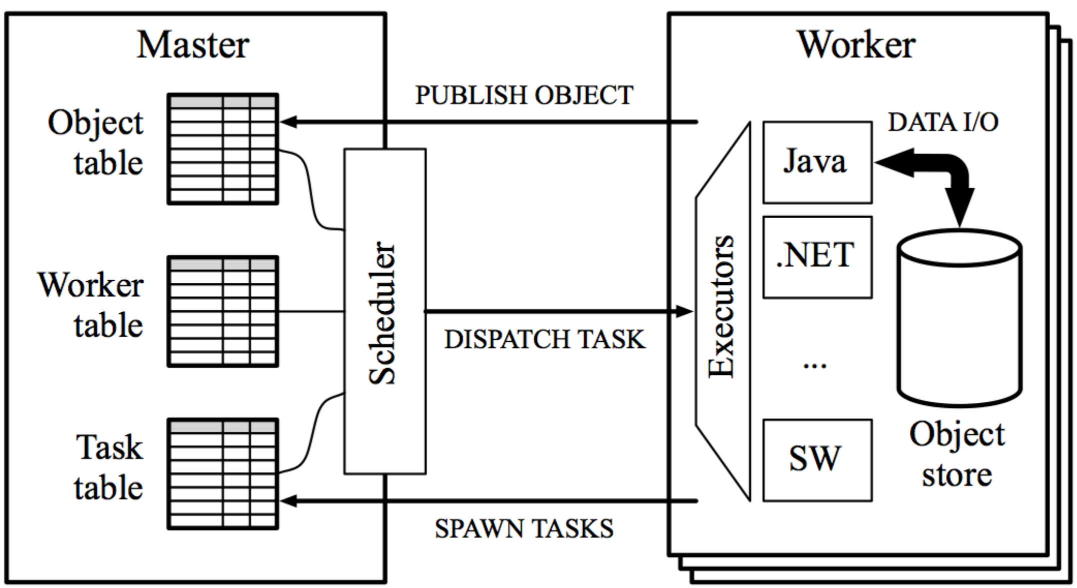
1. **Object**: An unstructured sequence of bytes (code, libraries, data, etc.)
2. **Reference**: The location where an object is stored
3. **Task**: A computation that executes completely on a single machine. Tasks can publish results and spawn other tasks.



An example task graph

System Architecture

- Master maintains current state of task graph in the object and task tables.
- Master does scheduling by lazily evaluating output objects, and pairs runnable tasks with idle workers.
- Workers execute tasks and store objects.



Skywriting

- A simple programming interface to CIEL

```
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;
```

Task Creation in Skywriting

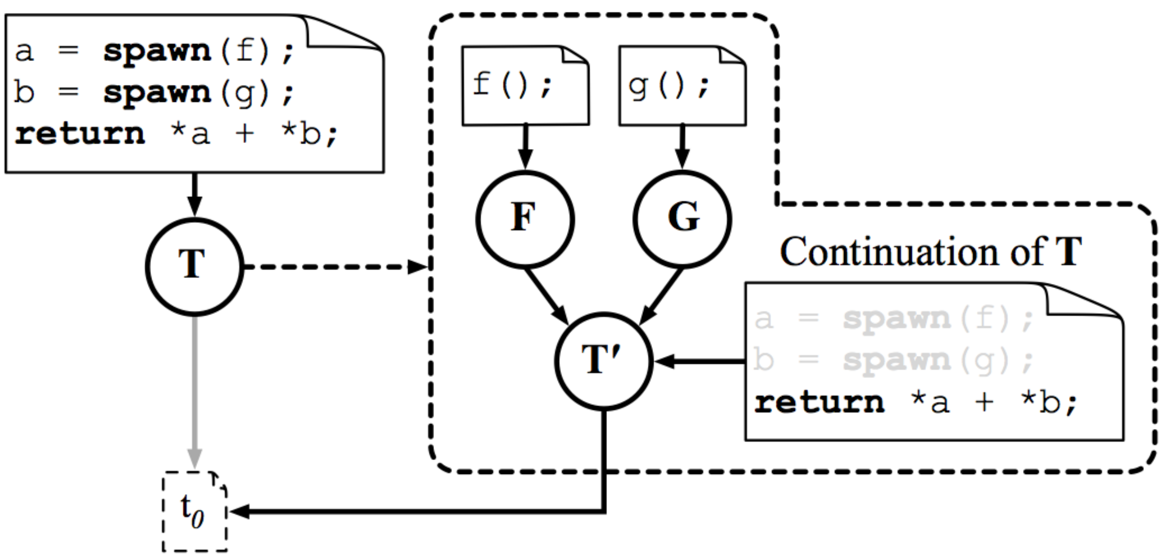
Task creation is the distinctive feature that facilitates data-dependent control flow. Two essential ways to create tasks in Skywriting:

1. spawn(f, args = [...])
   spawns a child task that computes and returns a pointer to f(args). Explicit task creation.
2. * (unary dereference operator that applies to a ref)
   Loads the referenced data and evaluates to the resulting data structure. Implicit task creation.

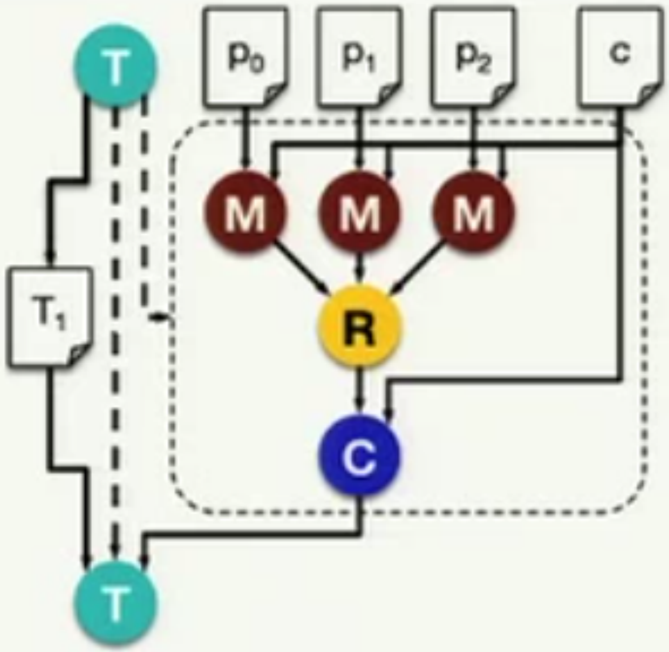
Implicit Task Creation with *

Problem: CIEL tasks are non-blocking, but dereferencing future objects will require waiting for tasks to complete.

Solution: Implicit creation of *continuation task*, which depends on dereferenced object and current execution stack.



Running a simple script




```
partitions = […];
guess = …;

do {
 prev = guess;
 guess = mapreduce(partitions,
                   lambda x: km_map(x, prev),
                   km_reduce);
 done = spawn(is_converged, [guess, prev]);
} while (!*done);
```

Outline

1. Motivation
2. Goals
3. Design
4. Fault Tolerance
5. Performance
6. Related Work
7. Conclusion

Fault Tolerance

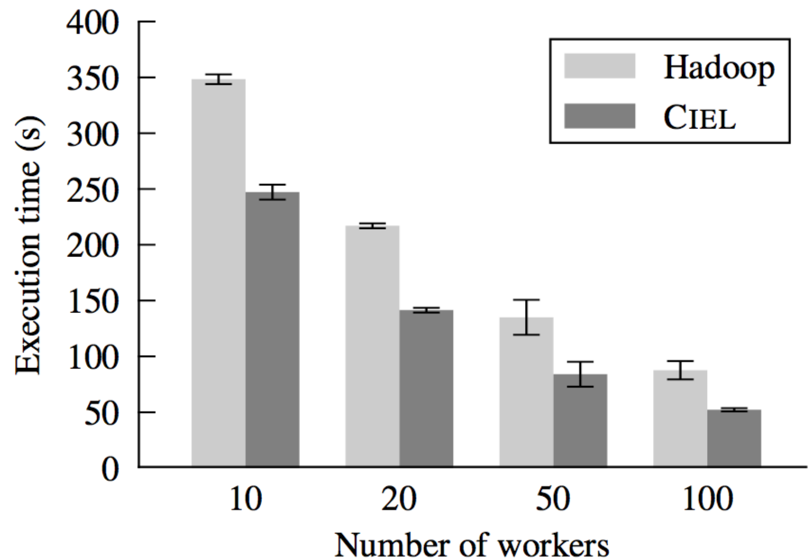
- **Client**: Trivial since no driver program is required.
- **Worker**: Monitored by master (similar to Dryad)
- **Master**: Master state can be derived from the set of active jobs. This is accomplished with
  - persistent logging, and
  - object table reconstruction by workers

Outline

1. Motivation
2. Goals
3. Design
4. Fault Tolerance
5. Performance
6. Related Work
7. Conclusion

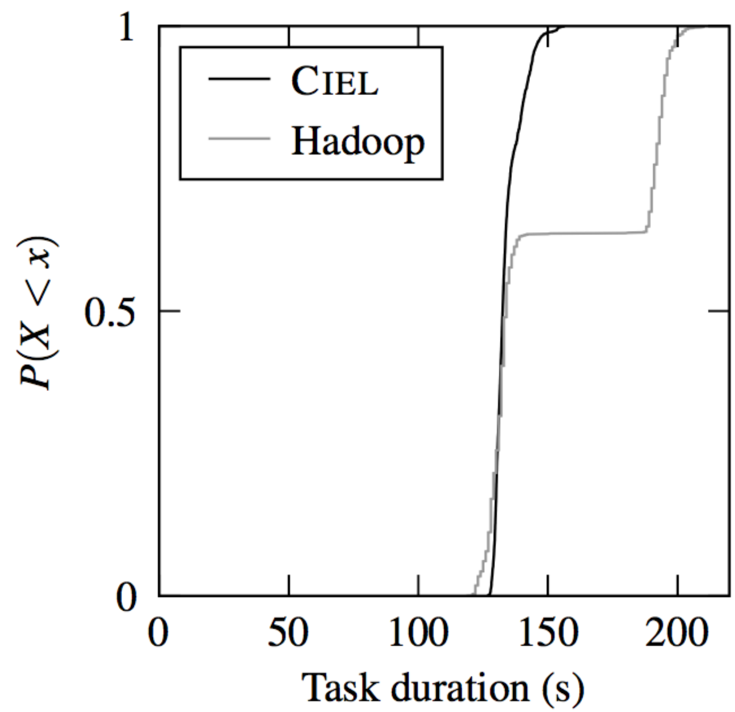
Experiment I: Grep

- How does CIEL compare to Hadoop?
- Hadoop polls for tasks once every 5 seconds. [this has changed since 2011. See patch: MAPREDUCE-1906]
- Hadoop runs mandatory “setup” and “cleanup” for each job
- Note **Hadoop’s weaker performance for small tasks.**



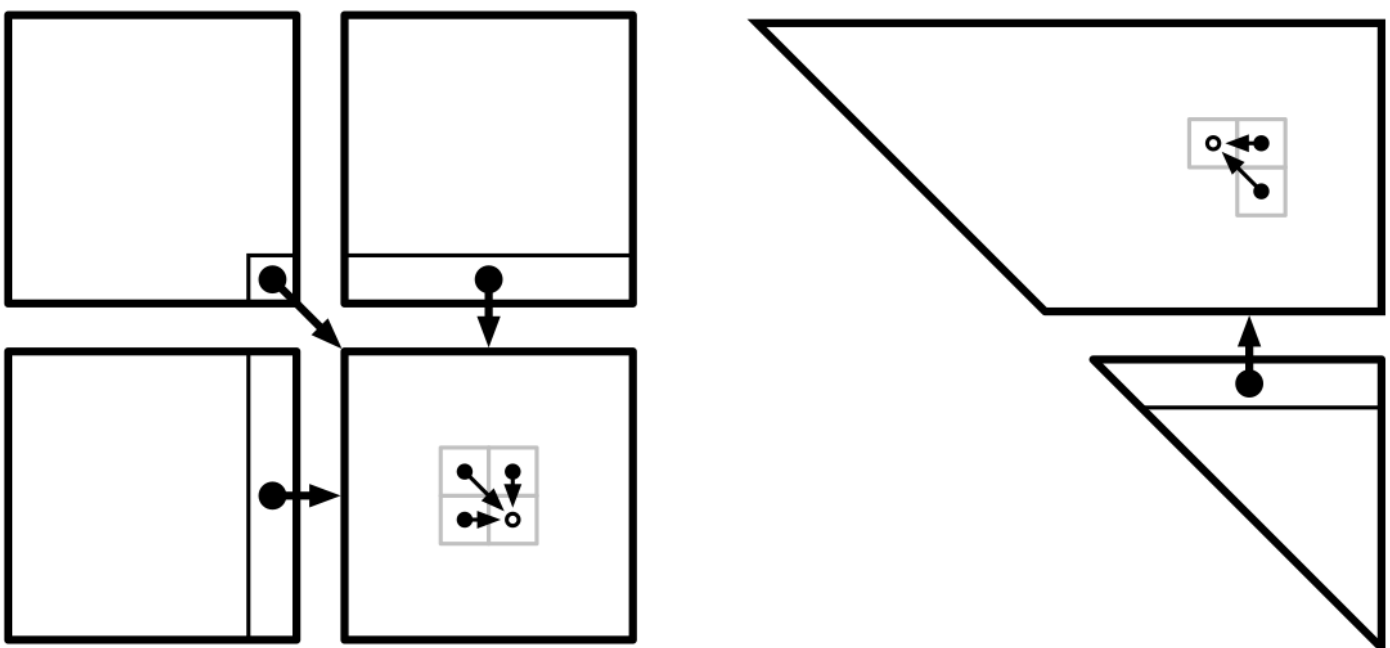
Experiment II: k-means

- How does CIEL compare to Hadoop (Apache Mahout) for iterative algorithms?
- Hadoop does not perform cross-job optimisations. Each iteration is an independent job.
- CIEL prefers workers that have consumed the same data for previous iterations, which leads to **better data-locality.**



Experiment III: DP

- CIEL can distribute partially parallelizable **tasks that do not cleanly fall into the MapReduce format**.



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

Goals (revisited)

Design a distributed execution framework that can

1. efficiently run iterative algorithms [dynamic task graph]
2. provide a simple interface [Skywriting]
3. offer transparent fault tolerance [Master fault tolerance]

Related Work

- Pregel: Google's distributed execution engine for graph algorithms [designed primarily for graph algorithms]
- HaLoop: task scheduler is made loop-aware by adding caching mechanisms [lacks fault tolerance]
- Apache Mahout: Uses Hadoop as its execution engine and a driver program runs iterative algorithms. [lacks master fault tolerance + requires driver program]

Conclusion

What are CIEL's significant contributions?

- Iterative Algorithms can be a single job. Therefore, there is no driver program running outside of the cluster.
- Dynamic Task Graph: Task spawns Task
- Fault tolerance for Master