CIEL
Control-flow in a distributed execution engine

Derek Murray
University of Cambridge
10th February 2011

Distributed execution is hard
1. Data and code distribution
2. Communication
3. Fault tolerance
4. Scheduling
Task parallelism

Distributed execution engines

1. Data and code distribution
2. Communication
3. Fault tolerance
4. Scheduling
Task dependencies/data-flow

MapReduce
Dryad

Jobs represented as:
• directed acyclic graphs
• static graphs

Distributed execution engines
1. Data and code distribution
2. Communication
3. Fault tolerance
4. Scheduling
5. Dependency tracking
Cyclic dependencies?

```
while (!converged) {
    // do stuff
}
```

Distributed execution engines

1. Data and code distribution
2. Communication
3. Fault tolerance
4. Scheduling
5. Dependency tracking
6. Control flow?
Dryad

Jobs represented as:
- directed **acyclic** graphs
- **static** graphs

Idea: dynamic task graphs

- Allow tasks to spawn other tasks

\[ \text{Input A} \rightarrow T \rightarrow \text{publishes} \rightarrow \text{Output X} \]

\[ \text{Input B} \rightarrow T \rightarrow \text{spawns} \]

\[ \text{M} \rightarrow \text{R} \rightarrow \text{G} \]
CIEL

- Execution engine for dynamic task graphs
- Supports various execution languages
  - Including *Skywriting* (later)
- Reliable execution on a distributed cluster
  - Client/master/worker fault tolerance (later)
Tasks and references

References
- Name objects
- Have GUIDs
- Are immutable
- May be concrete or future

Tasks
- Represent computation
  - Atomic
  - Non-blocking
  - Deterministic
- Depend on references
- Publish references
- Spawn tasks

CIEL architecture

Master

Tasks
Workers
References

Scheduler

Spawned tasks
Published references
Tasks to execute
CIEL architecture

Spawn → Publish

Worker

Java
.NET
Shell-script
Local objects

Defining dynamic task graphs

Input A
Input B

Output X

T

M

R

G
Skywriting

- Language for dynamic task graphs
  - Interpreted, dynamically-typed, C-like syntax
  - ...including a \texttt{while} statement

- Runs end-to-end on CIEL
  - One script, one job
  - Stored-program model
  - Fault tolerance throughout execution

Skywriting in a nutshell

\[ x = \texttt{spawn}(f); \]

- Future reference
- Function return
- Dereference operator
Blocking on futures

\[ x = \text{spawn}(f); \]
\[ y = *x; \]
\[ \text{return } y; \]
**k-means clustering**

![Diagram of k-means clustering](image)

**k-means in Skywriting**

```
p0 = ...; curr = ...;
do {
sums = [];
for (p in points) {
sums += spawn(km_map, [p, curr]);
}
old = curr;
curr = spawn(km_reduce, [sums]);
done = spawn(is_converged, [curr, old]);
} while (!done);
return curr;
```
**k-means on CIEL**

```
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];
    }
  }
  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;
}
```

MapReduce in Skywriting
Reliable execution on CIEL

- Any participant in the computation can fail

  - **Client** fault tolerance
    - Trivial due to whole-program execution

  - **Worker** fault tolerance
    - Re-execute tasks as necessary

  - **Master** fault tolerance

Task and reference naming

- Task (re-)execution must be **deterministic**

- In dynamic graph, how to choose names?
  - Deterministic function (SHA-1) of task inputs

- Lazy evaluation + deterministic naming
  - Task result memoization
Master fault tolerance

- Trivial version
  - Persistently store the root task for each job

- Better version

Implementation

- Implemented in ~8500 lines of Python
  - Plus executor code in Java, C, C#, ...

- Client/server based on JSON-RPC/HTTP
  - [http://github.com/mrry/ciel](http://github.com/mrry/ciel)
Applications

- Text-processing
  - Grep
  - Word count
- Clustering
  - $k$-means
- Link analysis
  - PageRank
- Linear algebra
  - Conjugate gradient
- Bioinformatics
  - Smith-Waterman
- Finance
  - Options pricing

Grep

<table>
<thead>
<tr>
<th>Workers</th>
<th>Hadoop</th>
<th>CIEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>350</td>
<td>340</td>
</tr>
<tr>
<td>20</td>
<td>250</td>
<td>245</td>
</tr>
<tr>
<td>50</td>
<td>150</td>
<td>145</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>95</td>
</tr>
</tbody>
</table>

Duration (seconds)
Grep

![Grep chart]

$k$-means

![$k$-means chart]
Binomial options pricing

Simulated time

Data flow

Binomial options pricing

Execution time (seconds)

Tasks

100000
200000
400000

2.2x faster
4.4x faster
6.0x faster
# Short-term future work

- More first-class languages on CIEL
  - Scala, Haskell, Java, Ocaml, …

- More platforms
  - Many-core and more exotic (e.g. Intel SCC)

- Hybrid multicore/distributed scheduling
  - Worker-local scheduler for lightweight tasks
  - Multi-scale concurrency interface

---

# Longer-term vision

- Rumors of SMP’s demise
  - Non-CC architectures (Beehive, SCC)
  - …or virtual machines in the cloud
  - …or both of the above

- How will we write applications in future?
  - Skywriting ≈ shell scripting
Conclusions

- Adding control-flow extends the class of programs that run on execution engines

- Skywriting lets you program in a simple imperative programming model

- CIEL achieves flexibility with competitive performance

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