CIEL

A universal execution engine for distributed data-flow computing

Murray, Derek G., et al. [1]

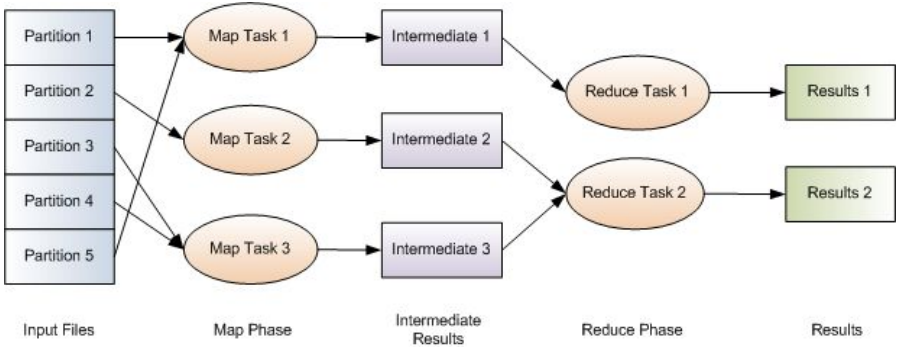
LSDPO (2017/2018) Paper Presentation
Ioana Bica (ib354)

Overview

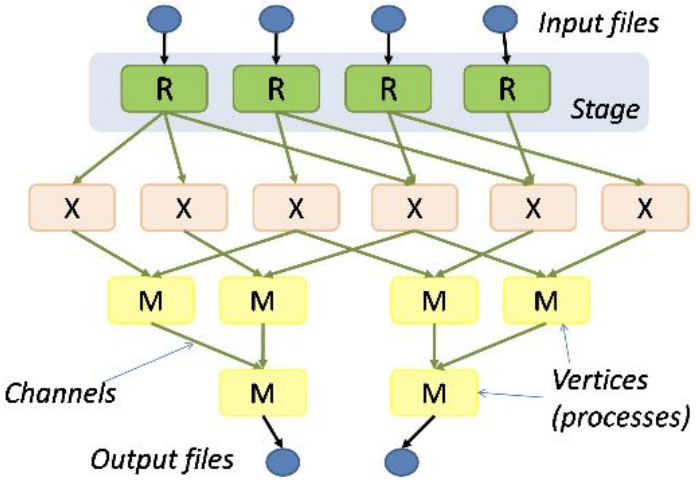
1. Motivation and related work
2. CIEL's contributions
3. Dynamic task graph and system architecture
4. Skywriting
5. Fault tolerance
6. Evaluation
7. Final remarks

Motivation

- Existing distributed execution engines (MapReduce and Dryad) were inefficient for iterative algorithms.



MapReduce job [2]

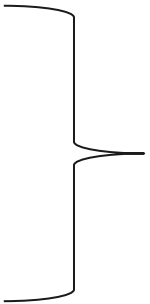


Dryad job [3]

Related work

Adding iteration capabilities to MapReduce:

- CGL-MapReduce
- HaLoop
- Apache Mahout



Do not provide transparent fault tolerance.

Do not support task dependency graphs.

Job latency is increased by consecutive iterations.

Related Work

Providing data-dependent control flow:

- Pregel

(Google's execution engine)

Composition of multiple computations not possible.

Only operates on a single dataset.

- Piccolo

(data-centric programming model)

Does not provide transparent scaling.

Fault tolerance involves checkpointing.

CIEL

- dynamic control flow
- dynamic task dependencies
- transparent fault tolerance
- transparent scaling
- data locality

Can execute iterative and recursive algorithms as a single job.

Contributions

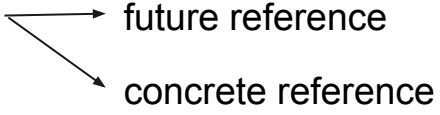
CIEL:

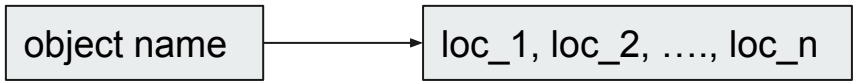
- dynamically builds a data-flow DAG as tasks execute
- increases the algorithmic expressibility in execution engines, by allowing iterative or recursive algorithms to be executed as a single job
- implements memoization of task results
- makes improvements to the fault tolerance mechanism

Dynamic task graph

Consists of the following CIEL primitives:

- objects
  - unstructured sequence of bytes
  - with unique name
- references

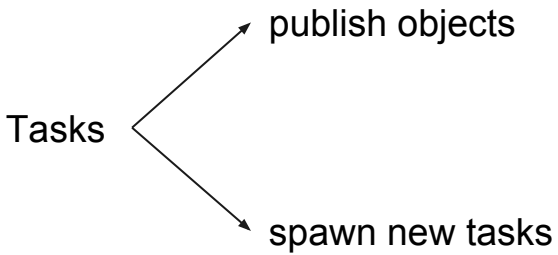




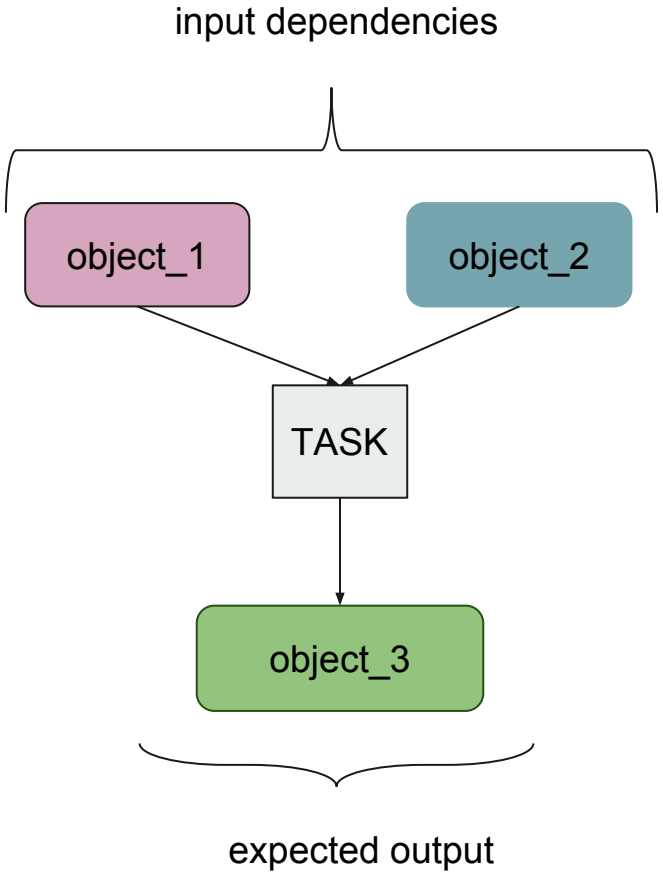
- tasks

Tasks

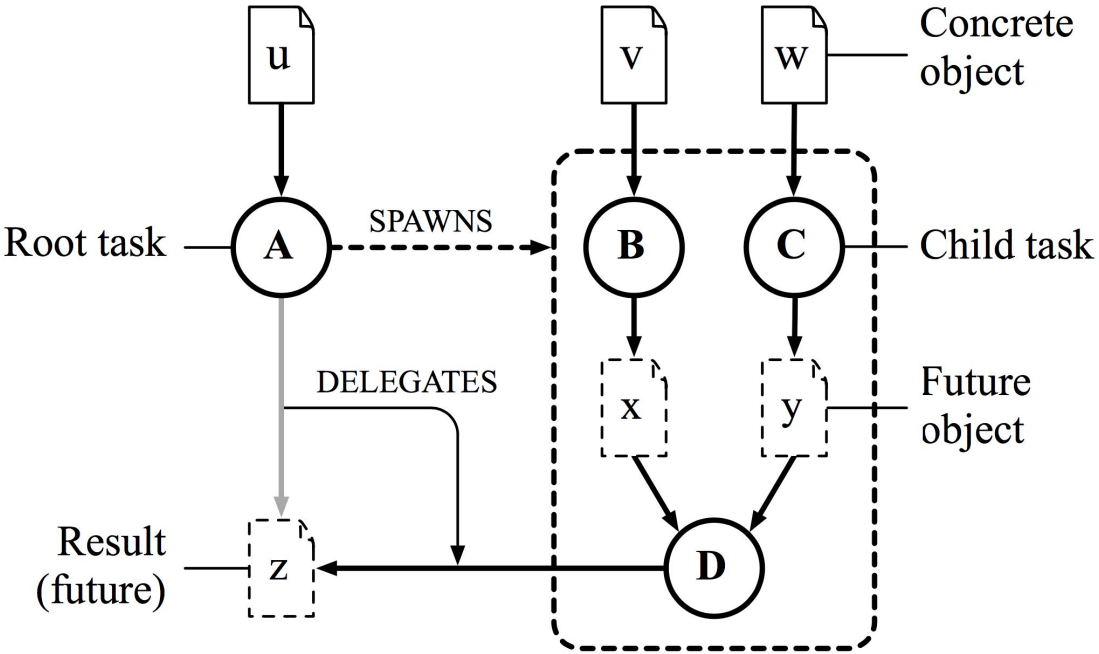
Non-blocking atomic computations.



Cycles cannot be formed in the dependency graph.

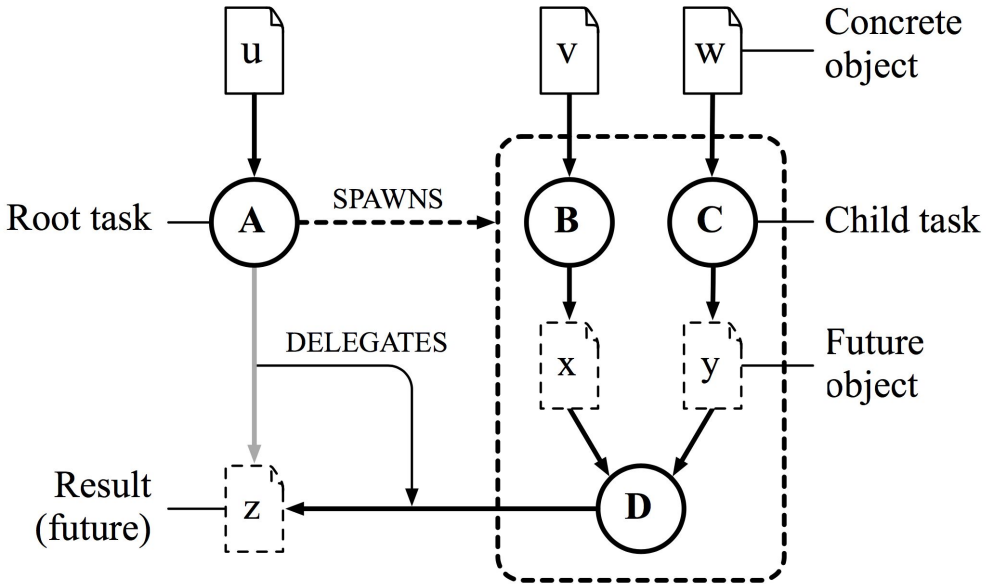


Dynamic task graph example



Lazy evaluation of objects

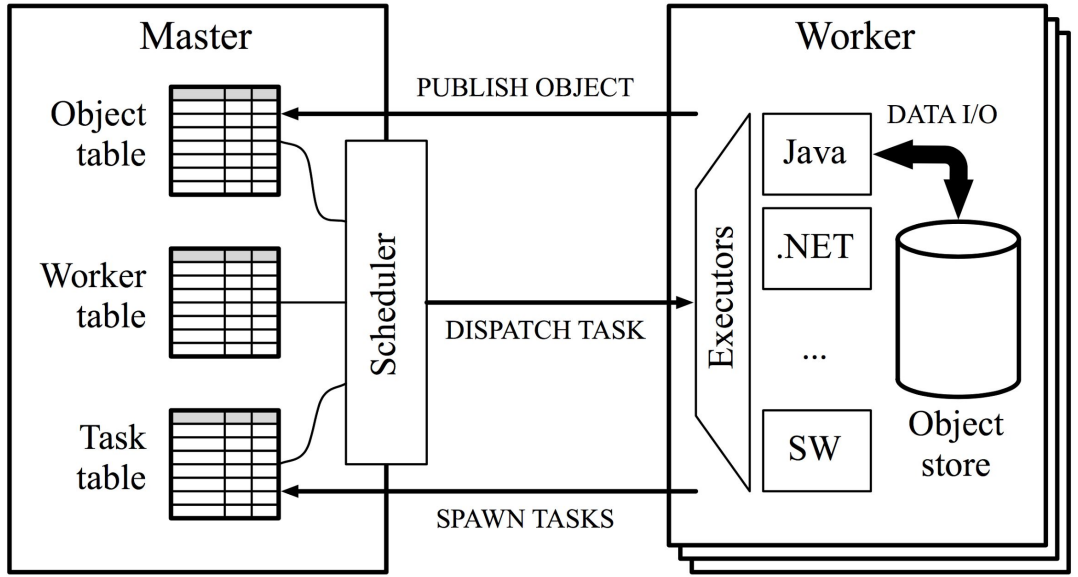
Start from the resulting object and recursively evaluate tasks as their dependencies become concrete.



System architecture

maintain current state of the dynamic task graph

keeps track of references published by tasks and the new spawned tasks



Tasks are dispatched to the worker nearest to the data.

Skywriting

- Turing complete programming language
- used to write parallelised jobs that can run on CIEL
- dynamically typed
- allows data mapping mechanisms through static file referencing

Skywriting can express arbitrary data-dependent control flow.

Key features

- `ref(url)`
- `spawn(f, [args, …])`
- `exec(executor, args, n)`
- `spawn_exec(executor, args, n)`
- `*` – dereference unary operator

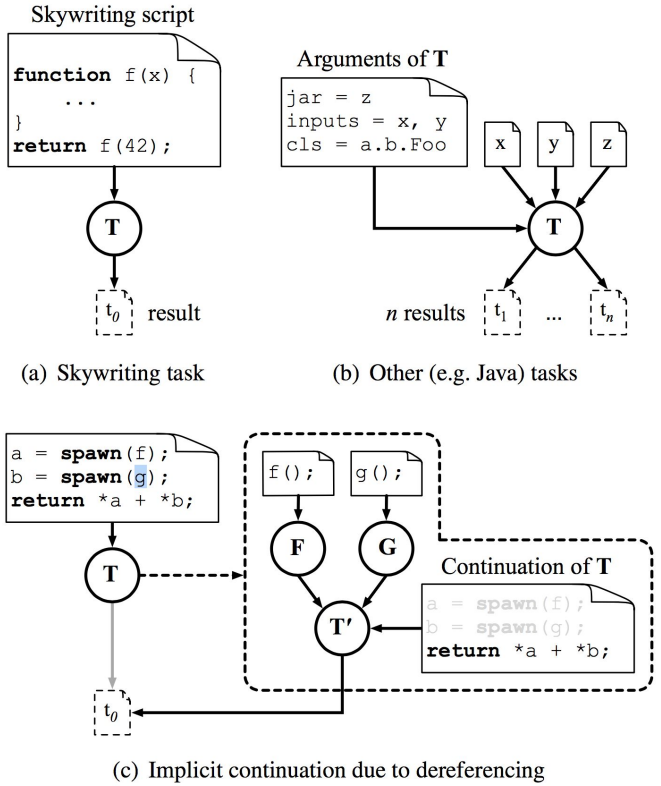
Using Skywriting to create tasks

Explicitly:

- using `spawn()` or `spawn_exec()`

Implicitly:

- using the `*`–operator



Memoisation

- memoise task results
- enabled by using deterministic naming for the objects:

executor	H(args||n)	i

- and by using lazy evaluation (only execute tasks if there outputs can resolve dependencies)

Fault tolerance

- Worker failures are handled similarly to Dryad
  - re-execute task performed by failed worker
  - re-execute tasks using data from the failed worker

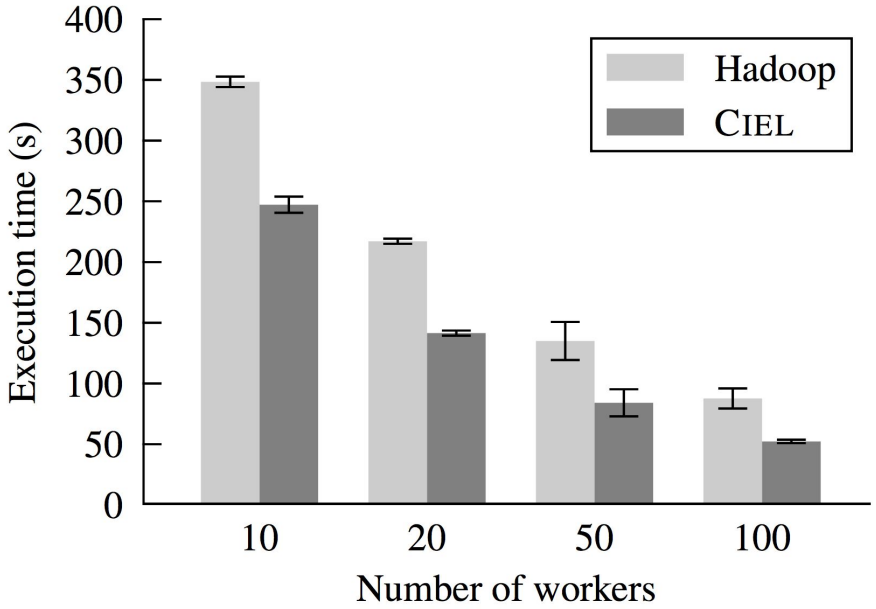
- Master failure: does not force the entire job to fail
  - derive master state from set of active jobs
  - use persistent logging and secondary masters

Evaluation

- grep benchmark
- k-means clustering
- dynamic programming
  - shows that CIEL has increased algorithmic expressivity compared to MapReduce
- impact of master failures on performance

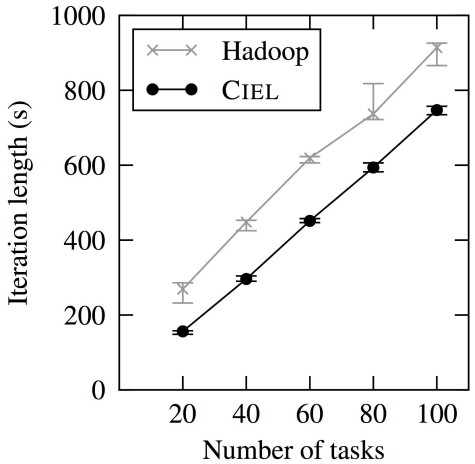
- No recursive algorithm?

Grep


400
350
300
250
200
150
100
50
0
Execution time (s)
Hadoop
CIEL
10
20
50
100
Number of workers

k-mean clustering

- CIEL achieves higher cluster utilization and less constant overhead
- CIEL is not any more scalable than Hadoop



When to use (or not) CIEL?

- CIEL enables clients to run iterative and recursive algorithms in a highly parallelized manner with transparent fault tolerance and transparent scaling

- CIEL was designed for coarse-grained parallelism across large data sets
  - For fine-grained parallelism, work-stealing schemes are better.
  - If data fits into RAM, Piccolo is more efficient.
  - If jobs share a lot of data, OpenMP is more appropriate.
  - For better scalability and performance use MPI.

Drawbacks and ideas for improvement

- CIEL does not control the number of tasks it spawns.
- Modifications to the data flow graph during execution are centralized.
- When a worker fails, all of the tasks that depend on the task executed by that worker need to be re-executed.

References

[1] Murray, Derek G., et al. "CIEL: a universal execution engine for distributed data-flow computing." *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 2011.

[2] www.cdmh.co.uk

[3] www.microsoft.com

[4] Dean, J., and S. Ghemawat. "MapReduce: simplified data processing on large clusters. OSDI'04 Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation", dalam: International Journal of Enggineering Science Invention." *URL: http://static. googleusercontent. com/media/resear ch. google. com (diunduh pada 2015-05-10)*(2004): 10-100.

[5] Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." *ACM SIGOPS operating systems review*. Vol. 41. No. 3. ACM, 2007.

Thank you!

Questions?