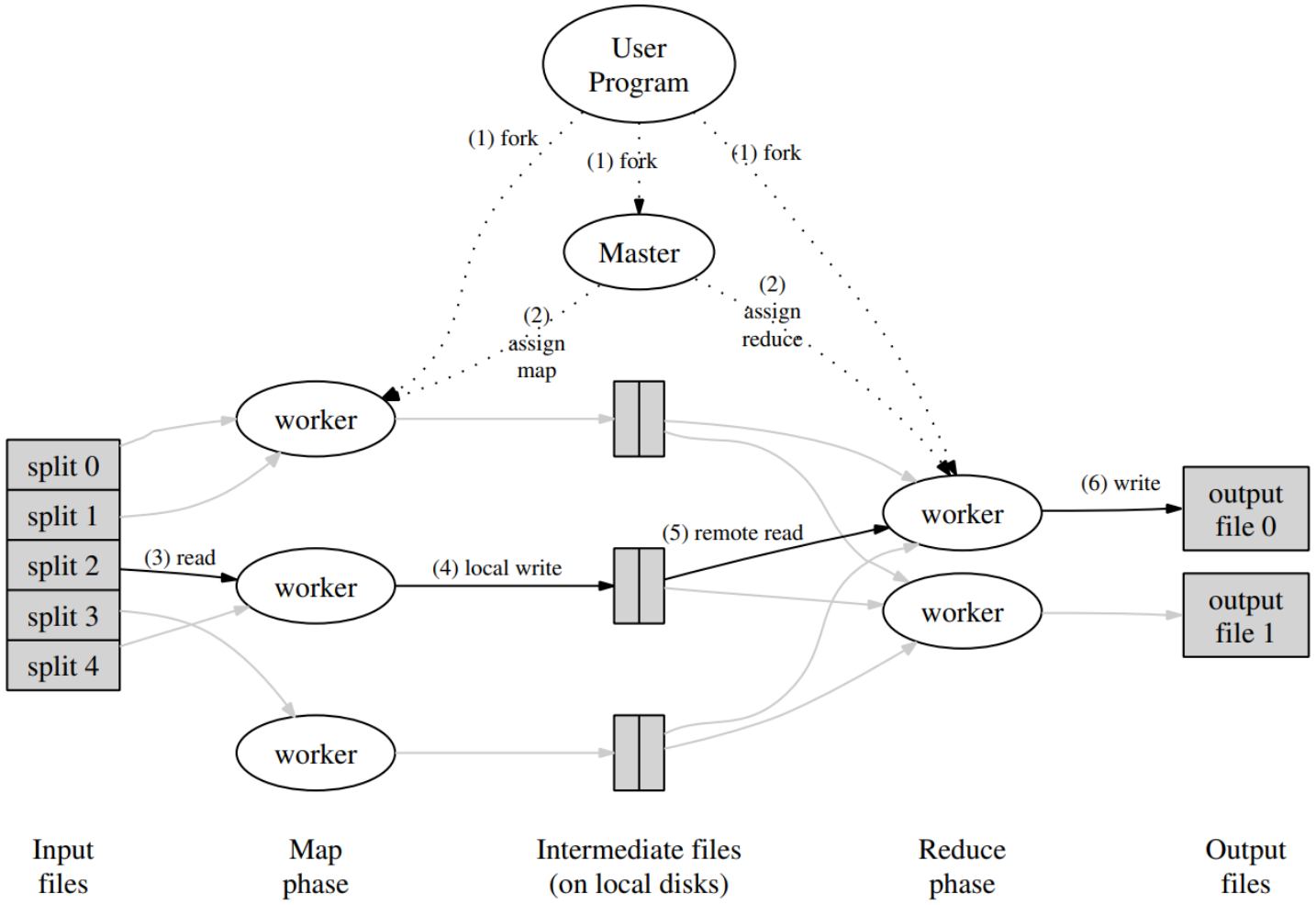
CIEL: a universal execution engine for distributed data-flow computing

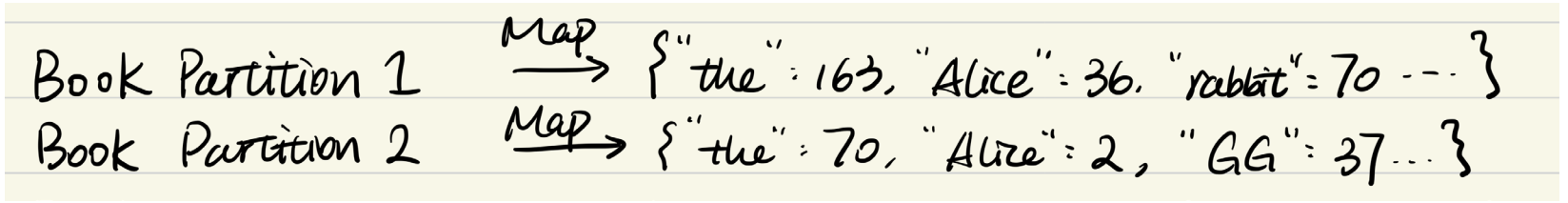
Research Review

MapReduce: a programming paradigm

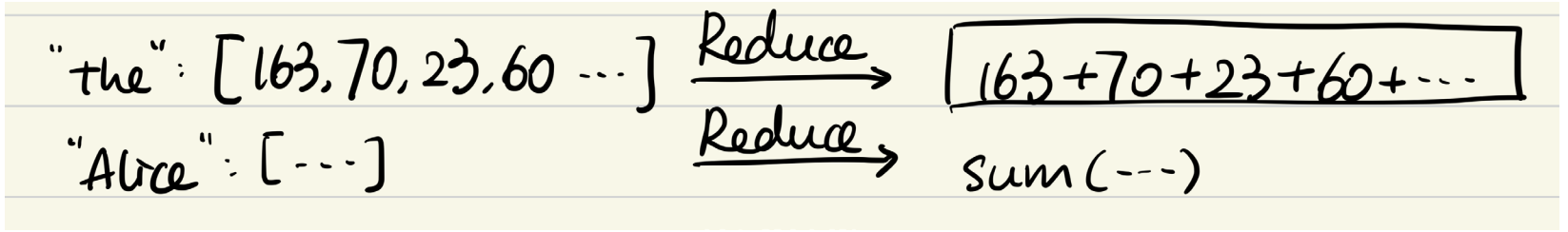


MapReduce example

- Task: summarize word occurrences in a thick book / database (100 TB)
- Split into chunks, assign to each worker
- Map: Each worker summarizes its own section



- Reduce: Add the number up



Merits

- Hadoop: an implementation
- The run-time system takes care of the details
  - partitioning the input data
  - scheduling the program’s execution across a set of machines
  - handling machine failures
  - managing the required inter-machine communication

Dryad

- Users pre-define a static data-flow graph (DAG)
- Dryad executes the vertices of the graph in some order

Issue: iterative algorithms

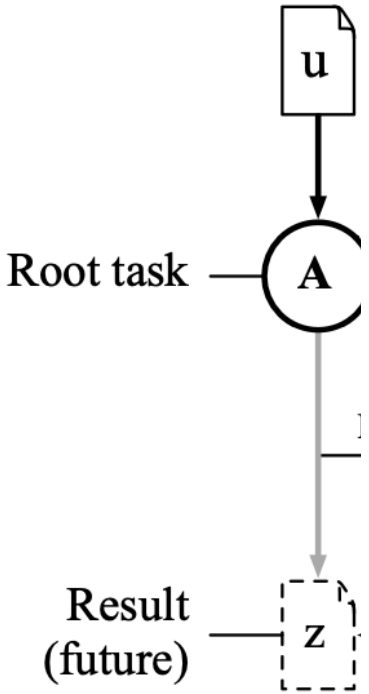
f(x) = f(x-1) + …

- **Latency**
  - MayReduce and Dryad are good for throughput, not latency
  - Overhead adds up for iterative tasks
- **Restricted control flow**
  - Data flow is limited: a bipartite graph / static DAG specified before the job
  - Cannot have “if previous job returns 1, execute this job”
  - Cannot achieve data-dependent (dynamic) control flow

Idea: Building a Dynamic DAG

- We need a more expressive **programming model** and more powerful **execution engine**

- **Tasks**: something to do, may have dependencies / expected output
- **Objects**: tasks input/output, immutable
- **References**: concrete (link to object) or future (object not produced)



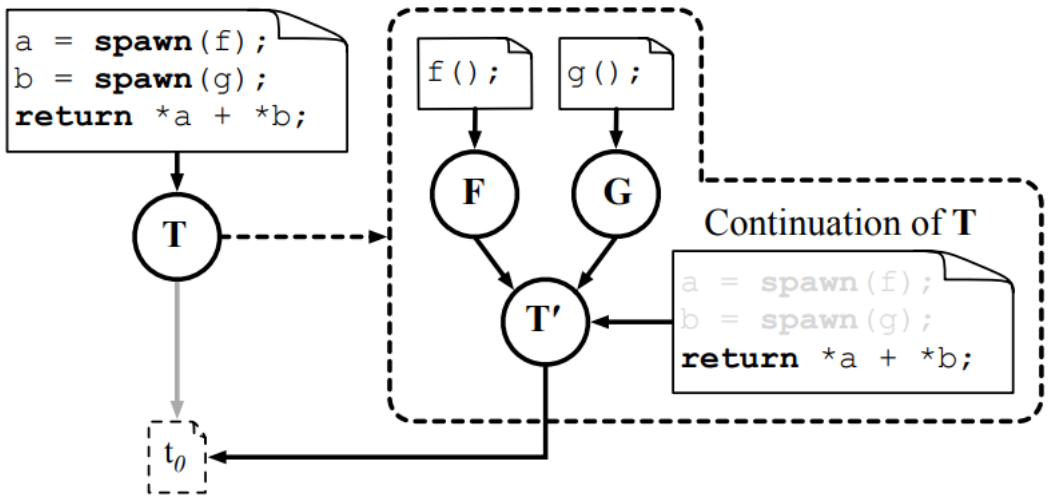
(a) Dynamic task graph

Turing-complete Scripting Language: SkyWriting

```
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;
```

Synchronization

- CIEL tasks are non-blocking
- all synchronization (and data-flow) must be made explicit in the dynamic task graph



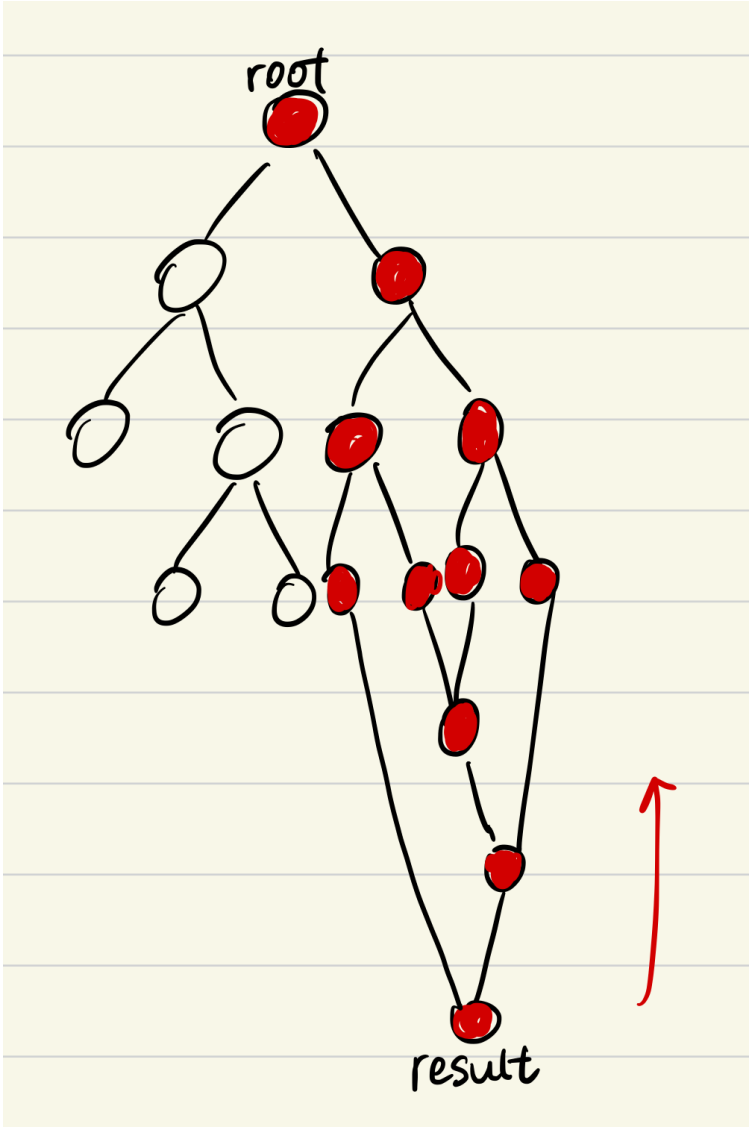
(c) Implicit continuation due to dereferencing

```
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 Execution

- **Clients**: set tasks
- **Master**: schedule tasks
- **Worker**: work on tasks

- **Lazy evaluation**: evaluate the output of root task until blocked



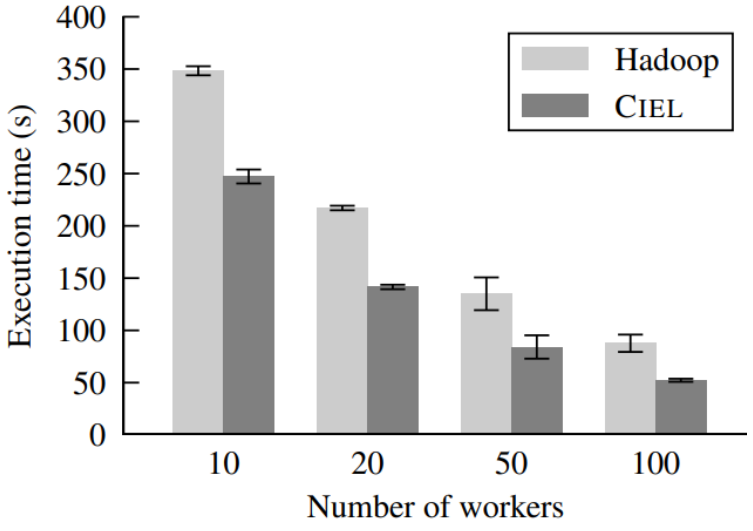
Evaluation: versatility

- MapReduce can be described in this language
- Many more tasks can be described in this paradigm
  - k-means algorithm
  - dynamic programming

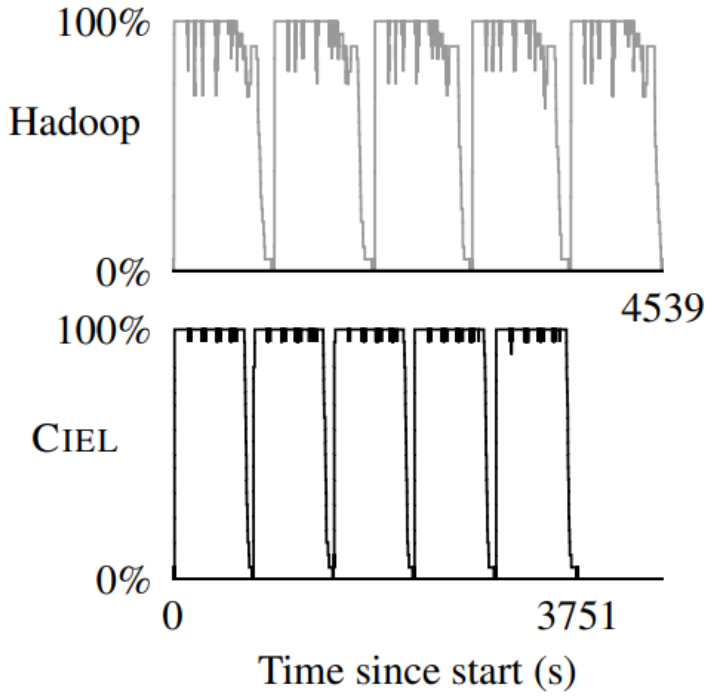
System Design: latency & utilization

- **Lazy evaluation**: good for latency, but throughput is traded
- **Streaming** results before fully saving on local disk
- Scheduler **considers object location** when assigning tasks
  - Tries to avoid inter-cluster communication
- Workers **not** using **polling** to get jobs

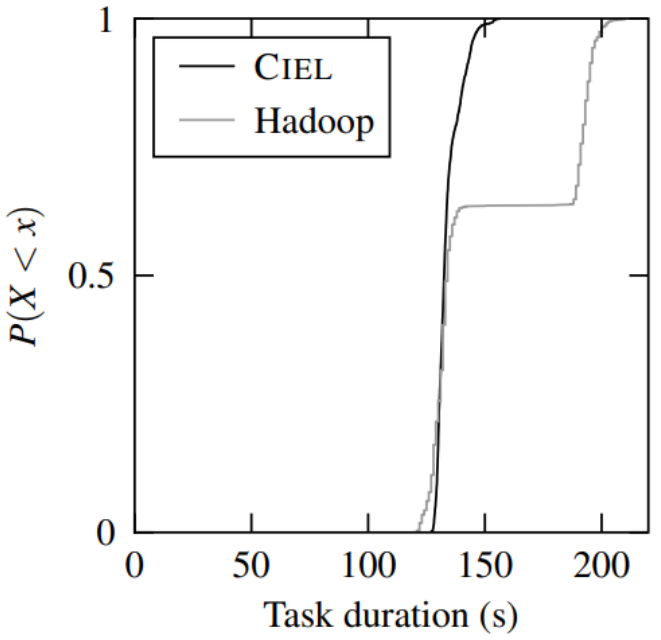
Evaluation: latency & utilization



Grep task: more workers, shorter task, greater % gain



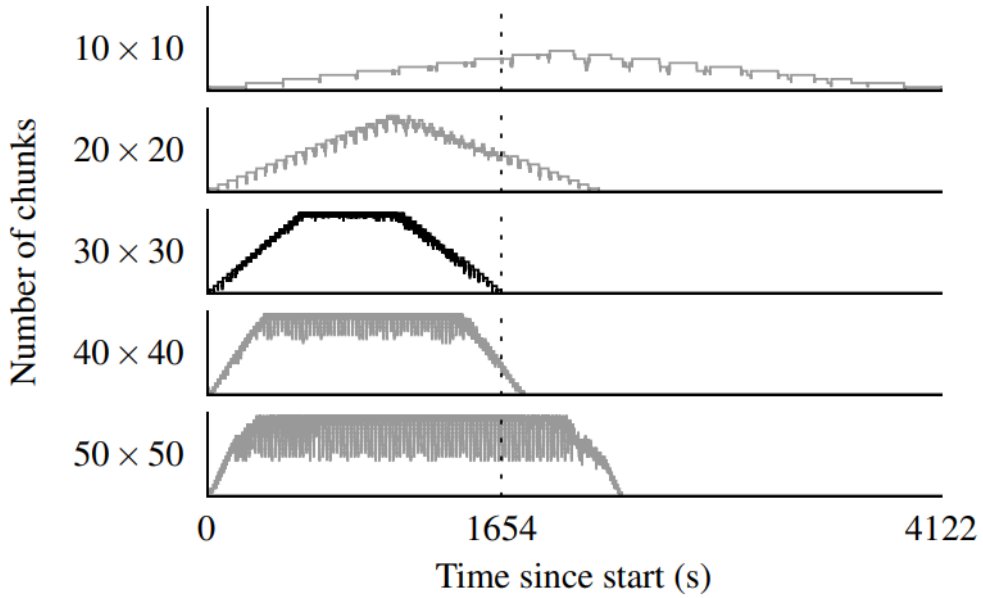
k-means task: higher utilization since polling is not used



k-means task: some Hadoop jobs are very slow due to non-local data

Task dispatch overhead

- If task dispatch is frequent, the overheads gets big
- Scheduler cannot dispatch tasks quickly enough
  - The system is written in Python 2 – that's the reason?



Smith-Waterman DP task: the utilization rate is noisy when problem size increases

Strengths and Weaknesses

- Strength:
  - Versatile data flow supported
  - Good short-task latency and utilization
  - Transparent fault tolerance
- Weaknesses:
  - Prototype software, not production-ready
  - Need to learn a new language
  - Scalability, job scheduler could be the bottleneck
  - Does not prioritize throughput
  - Coarse granularity: a whole machines considered as a single worker

Related Work

- OpenMP for HPC
  - Good for workers that share memory
- Adding keywords to existing languages
  - Cilk-NOW
- Declarative programming (with least fixed point operator)

How things have changed since then

- **CPU**: 1 (virtual) core / machine -> many cores
- **RAM**: 1.7G / machine -> much larger
- **Network** speed: 100 Mbps -> 100Gbps
- Machines more **stable**
- **Python** is still interpreted with GIL

- CIEL leaves multi-core utilization to the machine
  - There might be other systems that can handle that (Tensorflow?)
- Overhead of MapReduce might have been much better
- CIEL runtime is still slow

Current challenges for such distributed systems (as of 2021)

- Communication burden
- Straggler effect

Impact

- From citation counts, not so great compared to Dryad, MapReduce and TensorFlow
  - Other works have much better (industrial) support, while CIEL is more like a prototype
  - Problem overstated? Users prefer to handle data flow by themselves?
- Dynamic DAG very characteristic
- Improvement to latency well recognized