Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics

Omid Mashayekhi  Hang Qu  Chinmayee Shah  Philip Levis

July 13, 2017
Cloud Frameworks

Cloud frameworks abstract away the complexities of the cloud infrastructure from the application developers:

1. Automatic distribution
2. Elastic scalability
3. Multitenant applications
4. Load balancing
5. Fault tolerance
Cloud Frameworks

- **Job** is an instance of the application running in the framework.
- **Task** is the unit of computation for the job.
- **Control plane** partitions job in to tasks, schedules task, and recovers from faults.
Evolution of Cloud Frameworks

2004
I/O-bound data analytics
MapReduce
Hadoop

Task Length
10s  1s  100ms  10ms  1ms
Evolution of Cloud Frameworks

2004
- I/O-bound data analytics
- MapReduce
- Hadoop

2012
- In-memory data analytics
- Spark
- Naiad

Task Length
- 10s
- 1s
- 100ms
- 10ms
- 1ms
Evolution of Cloud Frameworks

- **2004**: I/O-bound data analytics
  - MapReduce
  - Hadoop
- **2012**: In-memory data analytics
  - Spark
  - Naiad
- **2016**: Optimized data analytics
  - Spark 2.0
  - Common IL
  - C++

**Task Length**
- 2004: 10s
- 2012: 1s
- 2016: 100ms
Individual tasks are getting faster.

But does it necessarily mean that job completion time is getting shorter?
Control Plane
The New Bottleneck

- Logistic regression over a data set of size 100GB.
- Classic Spark used to be **CPU-bound**.
Control Plane
The New Bottleneck

• Logistic regression over a data set of size 100GB.
• Spark 2.0 with Scala implementation is already \textbf{control-bound}.
Control Plane
The New Bottleneck

- Logistic regression over a data set of size 100GB.
- Spark-opt: hypothetical case where Spark runs tasks as fast as C++.
Control plane is the emerging bottleneck for the cloud computing frameworks.
Control Plane Design Scope

<table>
<thead>
<tr>
<th>Control Plane Design</th>
<th>Example Framework</th>
<th>Task Throughput (task per sec)</th>
<th>Scheduling Cost (per task)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>MapReduce, Hadoop, Spark</td>
<td>≈ 1,000</td>
<td>≈ 100 µs</td>
</tr>
<tr>
<td>Distributed</td>
<td>Naiad, TensorFlow</td>
<td>≈ 100,000</td>
<td>≈ 100,000 µs</td>
</tr>
</tbody>
</table>

- Centralized controller adapts to scheduling changes reactively with a low cost, but has limited task throughput and bottlenecks at scale.
- Distributed controller scales well, but any scheduling change requires stopping all nodes and installing new data flow with high latency.
Execution Templates is an abstraction for the control plane of cloud computing frameworks, that enables orders of magnitude higher task throughput, while keeping the fine-grained, flexible scheduling with low cost.
• Logistic regression over a data set of size 100GB.
• **Nimbus** with **execution templates** scales almost linearly, with low cost scheduling.
Repetitive Patterns

• Advanced data analytics are iterative in nature.
  – Machine learning, graph processing, image recognition, etc.

• This results in repetitive patterns in the control plane.
  – Similar tasks execute with minor differences.
Execution Model

Diagram showing the execution model with components labeled as:
- Driver Program
- Data
- Map
- Reduce
- Controller
- Task Graph
- Data Objects
- Worker

Data flow from Driver Program to Controller, then to Worker and Worker.
Execution Model

Driver Program
- Data
- Map
- Reduce

Controller
- Task Graph

Worker
- Data Objects

Worker
- Data Objects
Execution Model

Driver Program
- Data
- Map
- Reduce

Controller
- Task Graph

Worker
- Data Objects
- Task id
- Data list
- Dep. list
- Function
- Parameter

Worker
- Data Objects
Execution Model

Controller

Driver Program

Data Objects

Worker

Worker

Data Exchange

Task Graph

Data flow

Data

Map

Reduce

Task id
Data list
Dep. list
Function
Parameter
Repetitive Patterns

Controller

Task Graph

Data Objects

Worker

Data Objects

Worker
Repetitive Patterns

![Diagram of repetitive patterns with Controller, Task Graph, Worker, and Data Objects]

- **Controller** connects to **Task Graph**, which further connects to **Worker** and **Data Objects**.
- **Task id**, **Dep. list**, **Function**, and **Parameter** are highlighted in the diagram.

22
Repetitive Patterns

Controller

Task Graph

Worker

Data Objects

Data Exchange

Task id
Data list
Dep. list
Function
Parameter

Worker

Data Objects
Repetitive Patterns

Controller

Task Graph

Data Objects

C

Data list
Dep. list
Function
Parameter

Worker

Task id

Worker
Repetitive Patterns
Execution Templates

• Tasks are cached as parameterizable blocks on nodes.

• Instead of assigning the tasks from scratch, templates are instantiated by filling in only changing parameters.
Execution Templates

• Tasks are cached as **parameterizable blocks** on nodes.

• Instead of assigning the tasks from scratch, templates are **instantiated** by filling in only changing parameters.
Execution Templates
Mechanisms Summary

• **Instantiation**: spawn a block of tasks without processing each task individually from scratch. It helps increase the task throughput.

• **Edits**: modifies the content of each template at the granularity of tasks. It enables fine-grained, dynamic scheduling.

• **Patches**: In case the state of the worker does not match the preconditions of the template. It enables dynamic control flow.
Execution Templates

Instantiation

Controller

Task Graph

Data Objects

Worker

Data Objects

Worker
Execution Templates

Instantiation

Controller

Task Graph

Data Objects

Template

Worker

Data Objects

Template

Worker
Execution Templates

Instantiation
Execution Templates

Instantiation

Controller

Task Graph

Instantiate<params>

Data Objects

Worker

Template

Instantiate<params>

Data Objects

Template

Worker
Execution Templates

Instantiation

Controller

Task Graph

Worker

Data Objects

Template

Worker

Data Objects

Template
Execution Templates

Caching tasks implies static behavior; how could templates allow **dynamic scheduling**?

- Reactive scheduling changes for load balancing.
- Scheduling changes at the task granularity.
Execution Templates

Edits

• If scheduling changes, even slightly, the templates are obsolete.
  – For example rescheduling a task from one worker to another.

• Instead of paying the substantial cost of installing templates for every changes, templates allow edit, to change their structure.

• Edits enable adding or removing tasks from the template and modifying the template content, in-place.

• Controller has the general view of the task graph so it can update the dependencies properly, needed by the edits.
Execution Templates

Edits

Controller

Task Graph

Data Objects

Template

Worker

Data Objects

Template

Worker

Reschedule one task
Execution Templates

Edits

Controller

Task Graph

Edit<add>

Edit<remove>

Template

Data Objects

Worker

Template

Data Objects

Worker
Execution Templates

Edits

Controller

Task Graph

Data Objects

Template

Worker

Data Objects

Template

Worker

Template

38
Execution Templates

Edits

Controller

Task Graph

Instantiate<params>

Instantiate<params>

Worker

Worker

Data Objects

Data Objects

Template

Template
Execution Templates

Caching tasks implies static behavior; how could templates allow \textit{dynamic control flow}?

- Need to support nested loops.
- Need to support data dependent branches.
Execution Templates
Patching

• Execution templates operates at the granularity of **basic blocks**:
  – A code block with single entry and no branches except at the end.
• Each template has a set of **preconditions** that need to be satisfied.
  – For example the set of data objects in memory, accessed by the tasks.
• Worker state might not match the preconditions of the template in all circumstances.
• Controller **patches** the worker state before template instantiation, to satisfy the preconditions.
Execution Templates

Patching

Controller

Task Graph

Worker

Data Objects

Preconditions

Template

Worker

Data Objects

Preconditions

Template
Execution Templates

Patching

Controller

Task Graph

Worker

Preconditions

Template

Data Objects

Patch< load>

Worker

Preconditions

Template

Data Objects
Execution Templates

Patching

Controller

Task Graph

Preconditions

Data Objects

Worker

Template

Preconditions

Data Objects

Worker

Template
Execution Templates

Patching

Controller

Task Graph

Instantiate<params>

Worker

Preconditions

Data Objects

Template

Instantiate<params>

Worker

Preconditions

Data Objects

Template
Execution Templates

Patching

Controller

Task Graph

Worker

Worker

Preconditions

Data Objects

Template

Preconditions

Data Objects

Template

C
Execution Templates
Mechanisms Summary

• **Instantiation**: spawn a block of tasks without processing each task individually from scratch. It helps increase the **task throughput**.

• **Edits**: modifies the content of each template at the granularity of tasks. It enables fine-grained, **dynamic scheduling**.

• **Patches**: In case the state of the worker does not match the preconditions of the template. It enables **dynamic control flow**.
Nimbus

- Nimbus is designed for low latency, fast computations in the cloud.
- Nimbus embeds execution templates for its control plane.
- Nimbus supports traditional data analytics as well as Eulerian and hybrid graphical simulations; for the first time in a cloud framework.
  - Supervised/unsupervised learning algorithms.
  - Graph processing.
  - Physical simulation: water, smoke, etc. (PhysBAM library)
nimbus.stanford.edu

https://github.com/omidm/nimbus
Evaluation
Strong Scalability with Templates

- Logistic regression over data set of size 100GB.
- Spark-opt and Naiad-opt, runs tasks as fast as C++ implementation.
- Nimbus centralized controller with execution templates matches the performance of Naiad with a distributed control plane.
Evaluation
Reactive, Fine-Grained Scheduling with Templates

- Logistic regression over data set of size 100GB, on 100 workers.
- Naiad-opt curve is simulated (migrations every 5 iterations).
- Execution templates allow low cost, reactive scheduling changes.
  - Single edit overhead is only 41μs (in average).
Evaluation
High Task Throughput with Templates

- Spark and Nimbus both have centralized controller.
- Nimbus task throughput scales super linearly with more workers.
  - $O(N^2)$: more tasks and shorter tasks, simultaneously.
- For a task graphs with single stage:
  - Instantiation cost is <2μs per task (500,000 tasks per second).
Evaluation

Graphical Simulations Distributed in Nimbus

• To show the generality of execution templates, we considered graphical simulations in Nimbus:

  – Complex, and memory intensive from PhysBAM library.
  – High tasks throughput requirements (400,000 tasks per second).
  – Nested loops and data dependent branches.
  – Require patching in very subtle cases.
  – Traditionally in the HPC domain.
Evaluation

Graphical Simulations Distributed in Nimbus
## Conclusion

<table>
<thead>
<tr>
<th>Control Plane Design</th>
<th>Example Framework</th>
<th>Task Throughput (task per sec)</th>
<th>Scheduling Cost (per task)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>MapReduce, Hadoop, Spark</td>
<td>≈ 1,000</td>
<td>≈ 100μs</td>
</tr>
<tr>
<td>Distributed</td>
<td>Naiad, TensorFlow</td>
<td>≈ 100,000</td>
<td>≈ 100,000μs</td>
</tr>
<tr>
<td>Centralized w/ Execution Templates</td>
<td>Nimbus</td>
<td>≈ 100,000</td>
<td>≈ 100μs</td>
</tr>
</tbody>
</table>

Current cloud computing frameworks have either a centralized control plane model with fast, dynamic scheduling but limited task throughput, or a distributed control plane model with orders of magnitude higher task throughput but very high scheduling cost. Execution templates (§4) introduced by this dissertation enable Nimbus (§5) to achieve high task throughput while providing the fast, dynamic scheduling similar to centralized frameworks. This dissertation presents a third strategy using an abstraction called execution templates. Execution templates schedule at the same per-task granularity as centralized schedulers do. They do so while imposing the same minimal control overhead as distributed execution plans. Execution templates leverage the fact that long-running jobs (e.g. machine learning and graph processing) are iterative, running the same computation many times [119]. Machine learning algorithms, for example, typically iterate until the estimation error drops below a threshold.
Thank you!

nimbus.stanford.edu

https://github.com/omidm/nimbus