Fast, Distributed Computations in the Cloud

Omid Mashayekhi
Advisor: Philip Levis

April 7, 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
• **Job** is an instance of the application running in the framework.
• **Task** is the unit of computation.
• **Control plane** makes the magic happen:
  • Partitioning job in to tasks
  • Scheduling tasks
  • Load balancing
  • Fault recovery
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:
- 10s
- 1s
- 100ms
- 10ms
- 1ms
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++

10s 1s 100ms 10ms 1ms
Task Length
• One iteration of logistic regression over a data set of size 64MB.
• Tasks implemented efficiently, could run 50x faster.
Individual tasks are getting faster.

But does it necessarily mean that job completion time is getting shorter?
Cloud Frameworks

Task are getting orders of magnitude faster.

How about the job?
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 is already **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
The New Bottleneck

- Logistic regression over a data set of size 100GB.
- Nimbus with execution templates scales almost linearly.
Contributions

• Demonstrating how the control plane is the emerging bottleneck for data analytics frameworks.

• **Execution Templates** as 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.

• The design, implementation, and evaluation of **Nimbus**, a distributed cloud computing framework that embeds execution templates.

• A demonstration of a single-core **graphical simulation** that Nimbus automatically distributes in the cloud showing execution templates in practice for complex applications.
This talk

• Control Plane: the Emerging Bottleneck

• Design Scope of the Control Plane

• Execution Templates

• Nimbus: a Framework with Templates

• Evaluation
This talk

- Control Plane: the Emerging Bottleneck
- Design Scope of the Control Plane
- Execution Templates
- Nimbus: a Framework with Templates
- Evaluation
Cloud Frameworks Design

• Currently, there are two approaches:

  1. Centralized control model.
     • Controller generates and assigns tasks to the worker.
     • Limited task throughput, but reactive scheduling.

  2. Distributed data flow model.
     • Nodes generate and spawn tasks locally.
     • Great scalability, but static scheduling.
Centralized Controllers

- MapReduce
- Hadoop
- Spark

Distributed Controllers

- Controller centrally schedules and spawns tasks.
• Controller centrally schedules and spawns tasks.
Controller centrally schedules and spawns tasks.
• Controller could reactively and dynamically change the schedule.
• Controller could reactively and dynamically change the schedule.
Centralized Controllers

Distributed Controllers

- Controller could reactively and dynamically change the schedule.
Centralized Controllers

Distributed Controllers

- But controller bottlenecks at scale.
But controller bottlenecks at scale.
• But controller bottlenecks at scale.
• But controller bottlenecks at scale.
But controller bottlenecks at scale.
• Logistic regression over a data set of size 100GB in Spark 2.0 MLlib.
• Control Plane **bottlenecks at scale**, generating and spawning tasks.
• Each node generates and executes tasks locally.

Design Spectrum

Centralized Controllers

Distributed Controllers

• Naiad
• TensorFlow
• The design scales well as there is no single bottleneck.
- But, the scheduling is static.
  - The progress speed is bound to the speed of the slowest node.
  - Any change requires stopping all nodes and installing new data flow.
In practice the straggler mitigation is only proactive:
  • Avoiding stragglers by meticulous engineering work.
  • Launching backup workers (at least doubling the resources).
In practice the straggler mitigation is only **proactive**:
- Avoiding stragglers by meticulous engineering work.
- Launching backup workers (at least doubling the resources).
• In practice the straggler mitigation is only **proactive**:
  • Avoiding stragglers by meticulous engineering work.
  • Launching backup workers (at least doubling the resources).
## Design Space
### Summary

<table>
<thead>
<tr>
<th>Control Plane Design</th>
<th>Example Framework</th>
<th>Task Throughput</th>
<th>Task Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>MapReduce</td>
<td>Low</td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>Hadoop</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Spark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td>Naiad</td>
<td>High</td>
<td>Static</td>
</tr>
<tr>
<td></td>
<td>TensorFlow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We would like to have the best of both worlds:
- High task throughput for fast computations.
- Dynamic, fine-grained scheduling decisions.
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.
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.

```cpp
while (error > threshold_e) {
    while (gradient > threshold_g) {
        // Optimization code block
        gradient = Gradient(tdata, coeff, param)
        coeff += gradient
    }
    // Estimation code block
    error = Estimate(edata, coeff, param)
    param = update_model(param, error)
}
```
This talk

• Control Plane: the Emerging Bottleneck
• Design Scope of the Control Plane
  • Execution Templates
• Nimbus: a Framework with Templates
• Evaluation
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.

<table>
<thead>
<tr>
<th>Task id</th>
<th>Data list</th>
<th>Dep. list</th>
<th>Function</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task id</td>
<td>Data list</td>
<td>Dep. list</td>
<td>Function</td>
<td>Parameter</td>
</tr>
<tr>
<td>Task id</td>
<td>Data list</td>
<td>Dep. list</td>
<td>Function</td>
<td>Parameter</td>
</tr>
</tbody>
</table>
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.

![Diagram of execution templates]

<table>
<thead>
<tr>
<th>Task id</th>
<th>Data list</th>
<th>Dep. list</th>
<th>Function</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>P3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load New Task ids 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 Model

Driver Program

Data flow

- Data
- Map
- Reduce

Controller

Worker

Worker
Execution Model

Driver Program

Data flow

Data
Map
Reduce

Controller

Task Graph

Worker

Worker
Execution Model

- Driver Program
- Data flow
- Data
- Map
- Reduce

Controller
- Task Graph

Worker
- Data Objects
- Data Objects

Worker
Execution Model

Driver Program

Data flow
- Data
- Map
- Reduce

Controller

Task Graph

Worker
- Data Objects

Worker
- Data Objects
Execution Model

Driver Program
- Data
- Map
- Reduce

Controller
- Task Graph
  - Task id
  - Data list
  - Dep. list
  - Function
  - Parameter

Worker
- Data Objects

Worker
- Data Objects
Execution Model
Driver Program

while (error > threshold_e) {
    while (gradient > threshold_g) {
        // Optimization code block
        gradient = Gradient(tdata, coeff, param)
        coeff += gradient
    }
    // Estimation code block
    error = Estimate(edata, coeff, param)
    param = update_model(param, error)
}
Driver Program

```c
while (error > threshold_e) {
    while (gradient > threshold_g) {
        // Optimization code block
        gradient = Gradient(tdata, coeff, param)
        coeff += gradient
    }
    // Estimation code block
    error = Estimate(edata, coeff, param)
    param = update_model(param, error)
}
```
Repetitive Patterns

Driver Program

```
while (error > threshold_e) {
    while (gradient > threshold_g) {
        // Optimization code block
        gradient = Gradient(tdata, coeff, param)
        coeff += gradient
    }
    // Estimation code block
    error = Estimate(edata, coeff, param)
    param = update_model(param, error)
}
```

Controller

Task Graph

Data Objects

Worker

Data Exchange

Worker

Data Objects

Data list

Dep. list

Function

Parameter
Driver Program

```cpp
while (error > threshold_e) {
    while (gradient > threshold_g) {
        // Optimization code block
        gradient = Gradient(tdata, coeff, param)
        coeff += gradient
    }
    // Estimation code block
    error = Estimate(edata, coeff, param)
    param = update_model(param, error)
}
```
Repetitive Patterns

Driver Program

```plaintext
while (error > threshold_e) {
    while (gradient > threshold_g) {
        // Optimization code block
        gradient = Gradient(tdata, coeff, param)
        coeff += gradient
    }
    // Estimation code block
    error = Estimate(edata, coeff, param)
    param = update_model(param, error)
}
```
Execution Templates

Abstraction

Controller

Task Graph

Data Objects

Worker

Data Objects

Worker
Execution Templates

Abstraction

Controller

Task Graph

Worker

Data Objects

Template

Worker

Data Objects

Template
Execution Templates

Abstraction

Controller

Task Graph

Worker

Data Objects

Template

Worker

Data Objects

Template
Execution Templates

Abstraction

Controller

Task Graph

Instantiate<params>

Instantiate<params>

Worker

Worker

Data Objects

Data Objects

Template

Template

C
Execution Templates
Abstraction

Controller
Task Graph

Data Objects
Template
Worker

Data Objects
Template
Worker

63
Execution Templates
Abstraction

Controller

Task Graph

Worker

Data Objects

Template

Worker

Data Objects

Template

C
Execution Templates

_The Devil is in the details._

• Caching tasks implies static behavior:
  – Templates and **dynamic scheduling**?
    • Reactive scheduling changes for load balancing.
    • Scheduling changes at the task granularity.
  – Templates and **dynamic control flow**?
    • Need to support nested loops.
    • Need to support data dependent branches.
Execution Templates

*The Devil is in the details.*

- Caching tasks implies static behavior:
  - Templates and **dynamic scheduling**?
    - Reactive scheduling changes for load balancing.
    - Scheduling changes at the task granularity.
  - Templates and **dynamic control flow**?
    - Need to support nested loops.
    - Need to support data dependent branches.
Execution Templates
Edits

• If scheduling changes, even slightly, the templates are obsolete.
  – For example migrating tasks among workers.

• 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

Migrate one task

Worker
Execution Templates

Edits

Controller

Task Graph

Data Objects

Edit<add>

Edit<remove>

Worker

Data Objects

Template

Worker

Template

C
Execution Templates

Edits

Controller

Task Graph

Data Objects

Worker

Template

Data Objects

Worker

Template
Execution Templates
Edits

Controller

Task Graph

Instantiate<params>

Data Objects

Worker

Template

Instantiate<params>

Data Objects

Worker

Template

C

Template
Execution Templates

*The Devil is in the details.*

- Caching tasks implies static behavior:
  - Templates and *dynamic scheduling*?
    - Reactive scheduling changes for load balancing.
    - Scheduling changes at the task granularity.
  - Templates and *dynamic control flow*?
    - Need to support nested loops.
    - Need to support data dependent branches.
Execution Templates
Granularity
• The more tasks cached in the template the better.
  – The cost of template instantiation is amortized over greater number of tasks.
  – But **loop unrolling** only works for static control flow.
Execution Templates
Granularity
Execution Templates
Granularity
Execution Templates
Granularity

- Cannot reuse the template (only two iterations of the inner loop).
• Templates cannot go beyond a branch in the driver program.

• Execution templates operates at the granularity of **basic blocks**:  
  – A code block with single entry and no branches except at the end.  
  – It is the biggest block without sacrificing **dynamic control flow**.
Execution Templates
Granularity

Template 1
Execution Templates
Granularity
Execution Templates
Granularity

[Diagram of iterative optimization process with nodes labeled as Training Data, Estimation Data, Coefficients, Parameters, and Iterative Optimizer.]

Template 2
Execution Templates

Granularity
Execution Templates
Granularity

Controller

Task Graph

EndTemplate

StartTemplate

Worker

Data Objects

Worker

EndTemplate

StartTemplate

Data Objects
Execution Templates

Granularity

Controller

Task Graph

Worker

Template

Data Objects

Worker

Template

Data Objects

Template

C
Execution Templates

Patching

- With dynamic control flow a basic block can have different entries.
- The execution state is not similar in all circumstances.
Execution Templates

Patching

Diagram:
- Training Data
- Estimation Data
- Coefficients
- Parameters
- Iterative Optimizer
- Error Estimation
- Instantiate Template 1
- Instantiate Template 1
Execution Templates

Patching

Diagram:
- Training Data
- Estimation Data
- Iterative Optimizer
- Error Estimation
- Coefficients
- Parameters

Steps:
1. Instantiate Template 1
2. Estimation
3. Error Estimation
4. Instantiate Template 1
5. Iterate

Note:
- Instan2ate Template 1
Execution Templates

Patching

- Training Data
- Estimation
- Parameters
- Error Estimation
- Iterative Optimizer

Instantiate Template 1

Instantiate Template 1
Execution Templates

Patching

Updated model parameters only on the reducer
Execution Templates  
Patching

• Each template has a set of **preconditions** that need to be satisfied before it can be instantiated.

  – For example the set of data objects in memory, accessed by the tasks cached in the template.
Execution Templates

Patching

Diagram:
- Controller
- Task Graph
- Worker
  - Data Objects
  - Template
- Worker
  - Data Objects
  - Template
  - C

Note: Diagram illustrates the execution of templates with patching, showing the flow from the controller to the task graph and then to the workers.
Execution Templates
Patching

Controller
Task Graph

Worker
Template
Preconditions
Data Objects

Worker
Template
Preconditions
Data Objects

Preconditions
Data Objects
Execution Templates

Patching

• Each template has a set of **preconditions** that need to be satisfied before it can be instantiated.
  
  – For example the set of data objects in memory, accessed by the tasks cached in the template.

• 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

Preconditions

Data Objects

Template

Worker

Preconditions

Data Objects

Template
Execution Templates

Patching

Controller

Task Graph

Preconditions

Data Objects

Template

Worker

Patch< load >

Preconditions

Data Objects

Template

Worker
Execution Templates

Patching

Controller

Task Graph

Worker

Preconditions

Data Objects

Template

Worker

Preconditions

Data Objects

Template

C
Execution Templates

Patching

Controller

Task Graph

Instantiate<params>

Preconditions

Data Objects

Worker

Instantiate<params>

Preconditions

Data Objects

Worker

Template

C
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**.
This talk

• Control Plane: the Emerging Bottleneck

• Design Scope of the Control Plane

• Execution Templates

• **Nimbus: a Framework with Templates**

• Evaluation
Nimbus

• Nimbus is designed for low latency, fast computations in the cloud.
  – Implemented in C++ (the core library is ~35,000 semicolons).
  – Mutable data model to allow in-place operations.

• Nimbus embeds execution templates for its control plane.
  – The centralized controller allows dynamic scheduling and resource allocation.
  – Execution templates help deliver high task throughput at scale.

• 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 library.
  – PhysBAM library (water, smoke, etc.)
Nimbus
Control Flow

- Tasks spawn other tasks for execution (similar to Legion).
- Driver program is a lineage of tasks executing on the workers.
- More flexible DAG for the task graph.
  - Not just narrow and wide dependencies.
  - Needed for graphical simulations.
Nimbus
Controller and Worker Templates

Controller

Worker
Worker
Worker
Worker

Instantiate

Controller Templates
Nimbus
Controller and Worker Templates

Controller

Worker Templates

Worker  Worker  Worker  Worker
Nimbus
Controller and Worker Templates

Controller

Worker
Worker
Worker
Worker

Worker Templates
Nimbus
Controller and Worker Templates

Controller

Worker
Worker
Worker
Worker

Worker Templates
• The goal is to automatically distribute sequential library kernels.
• Four layer data abstraction (geometric, logical, physical, application).
• Automatic translation and caching between the data layers.
• For more information you can visit Nimbus website.
This talk

- Control Plane: the Emerging Bottleneck
- Design Scope of the Control Plane
- Execution Templates
- Nimbus: a Framework with Templates
- Evaluation
Evaluation
Results Summary

• Control plane task throughput:
  – Execution templates match the strong scaling performance of frameworks with distributed control plane design.

• Dynamic scheduling:
  – Execution templates allows low cost, reactive scheduling and dynamic resource allocation similar to a centralized frameworks.

• Dynamic control flow:
  – Execution templates can handle applications with nested loops and data dependent branches with low overhead.
**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.
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 through edits at task granularity.

- Single edit overhead is only 41μs (in average).
Evaluation
Dynamic Resource Allocation with Templates

- Logistic regression over 100GB of data, on 50/100 workers.
- One-time template installation cost is ~40% of direct task scheduling.
- Nimbus allows dynamic resource allocation.
- Nimbus installs multiple versions of a template depending on resources.
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
Evaluation
Complexities of Graphical Simulations

- 40 different variables: scalar, vector, particle.
- Triply nested loop with data dependent branches.
  - 9 different templates (basic blocks).
  - 3 branches that need patching.
• Canonical water simulations under Nimbus and MPI.
• Without templates, Nimbus is almost 6x slower than MPI.
• Slow down means either lower resolution or more time/money.
Evaluation
Speedup with Templates

• Canonical water simulations under Nimbus and MPI.
• Without templates, Nimbus is almost 6x slower than MPI.
• Slow down means either lower resolution or more time/money.
• Canonical water simulations under Nimbus and MPI.
• Nimbus performance is within 3-15% of the hand-tuned MPI.
• At 512 cores, there are more than 1 million distinct data objects and task throughput picks at 460,000 tasks per second.
Evaluation
Load Balancing and Fault Recovery with Templates

- Nimbus controller adapts to the stragglers and worker failures.
- Templates are seamlessly installed as schedule changes.
Contributions

• Demonstrating how the control plane is the emerging bottleneck for data analytics frameworks.

• Execution Templates as 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.

• The design, implementation, and evaluation of Nimbus, a distributed cloud computing framework that embeds execution templates.

• A demonstration of a single-core graphical simulation that Nimbus automatically distributes in the cloud showing execution templates in practice for complex applications.
## Conclusion

<table>
<thead>
<tr>
<th>Control Plane Design</th>
<th>Example Framework</th>
<th>Task Throughput</th>
<th>Task Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>MapReduce</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hadoop</td>
<td>Low</td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>Spark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td>Naiad</td>
<td>High</td>
<td>Static</td>
</tr>
<tr>
<td></td>
<td>TensorFlow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centralized w/</td>
<td>Nimbus</td>
<td>High</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Execution Templates</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thank You!