Starling: A Scheduler Architecture for High Performance Cloud Computing

Hang Qu, Omid Mashayekhi, David Terei, Philip Levis
Stanford Platform Lab Seminar
May 17, 2016
High Performance Computing (HPC)

• Arithmetically dense codes (CPU-bound)
• Operate on in-memory data

SpaceX  NOAA  Pixar

• A lot of machine learning is HPC
HPC on the Cloud

• HPC has struggled to use cloud resources

• HPC software has certain expectations
  – Cores are statically allocated, never change
  – Cores do not fail
  – Every core has identical performance

• Cloud doesn’t meet these expectations
  – Elastic resources, dynamically reprovisioned
  – Expect some failures
  – Variable performance
HPC in a Cloud Framework?

• Strawman: run HPC codes inside a cloud framework (e.g., Spark, MapReduce, Naiad, etc.)
  – Framework handles all of the cloud’s challenges for you: scheduling, failure, resource adaptation

• Problem: too slow (by orders of magnitude)
  – Spark can schedule 2,500 tasks/second
  – Typical HPC compute task is 10ms
    ‣ Each core can execute 100 tasks/second
    ‣ A single 18 core machine can execute 1,800 tasks/second
  – Queueing theory and batch processing mean you want to operate well below the maximum scheduling throughput
Starling

• Scheduling architecture for high performance cloud computing (HPCC)

• Controller decides data distribution, workers decide what tasks to execute
  – In steady state, no worker/controller communication except periodic performance updates

• Can schedule up to 120,000,000 tasks/s
  – Scales linearly with number of cores

• HPC benchmarks run 2.4-3.3x faster
Outline

• HPC background
• Starling scheduling architecture
• Evaluation
• Thoughts and questions
High Performance Computing

• Arithmetically intense floating point problems
  – Simulations
  – Image processing
  – If you might want to do it on a GPU but it’s not graphics, it’s probably HPC

• Embedded in a geometry, data dependencies
Example: Semi-lagrangian Advection

A fluid is represented as a grid
Each cell has a velocity
Update value/velocity for next time step
Example: Semi-lagrangian Advection

A fluid is represented as a grid
Each cell has a velocity
Update value/velocity for next time step
Example: Semi-lagrangian Advection

A fluid is represented as a grid
Each cell has a velocity
Update value/velocity for next time step

Walk velocity vector backwards, interpolate
Example: Semi-lagrangian Advection

A fluid is represented as a grid
Each cell has a velocity
Update value/velocity for next time step

Walk velocity vector backwards, interpolate

What if partitions A+B are on different nodes?
Example: Semi-lagrangian Advection

A fluid is represented as a grid.
Each cell has a velocity.
Update value/velocity for next time step.

Walk velocity vector backwards, interpolate.

What if partitions A+B are on different nodes?

Each partition has partial data from other, adjacent partitions ("ghost cells").
High Performance Computing

• Arithmetically intense floating point problems
  – Simulations
  – Image processing
  – If you might want to do it on a GPU but it’s not graphics, it’s probably HPC

• Embedded in a geometry, data dependencies
  – Many small data exchanges after computational steps
  – Ghost cells, reductions, etc.
Most HPC Today (MPI)

while (time < duration) {
    // locally calculate, then global min
    dt = calculate_dt();
    // locally calculate, then exchange
    update_velocity(dt);
    // locally calculate, then exchange
    update_fluid(dt);
    // locally calculate, then exchange
    update_particles(dt);
    time += dt;
}

Control flow is explicit in each process.
Operate in lockstep.
Partitioning and communication is static.
A single process fails, program crashes.
Runs as fast as slowest process.
Assumes every node runs at the same speed.

Problems in the cloud
Most HPC Today (MPI)

```c
while (time < duration) {
    // locally calculate, then global min
    dt = calculate_dt();
    // locally calculate, then exchange
    update_velocity(dt);
    // locally calculate, then exchange
    update_fluid(dt);
    // locally calculate, then exchange
    update_particles(dt);
    time += dt;
}
```
Control flow is explicit in each process.
Operate in lockstep.
Partitioning and communication is static.
A single process fails, program crashes.
Runs as fast as slowest process.
Assumes every node runs at the same speed.
HPC inside a cloud framework?

![Diagram showing HPC code running on cloud (EC2, GCP) within a cloud framework that provides load balancing, failure recovery, and elastic resources for stragglers.]
Frameworks too slow

• Depend on a central controller to schedule tasks to worker nodes
• Controller can schedule ~2,500 tasks/second
• Fast, optimized HPC tasks can be ~10ms long
  – 18-core worker executes 1,800 tasks/second
  – Two workers overwhelm a scheduler
  – Prior work is fast or distributed, not both
Outline

• HPC background
• Starling scheduling architecture
• Evaluation
• Thoughts and questions
Best of both worlds?

• **Depend on a central controller** to
  – Balance load
  – Recover from failures
  – Adapt to changing resources
  – Handle stragglers

• **Depend on worker nodes** to
  – Schedule computations
    ‣ Don’t execute in lockstep
    ‣ Use AMT/software processor model (track dependencies)
  – Exchange data
Control Signal

• Need a different control signal between controller and workers to schedule tasks
• Intuition: tasks execute where data is resident
  – Data moves only for load balancing/recovery
• Controller determines how data is distributed
• Workers generate schedule tasks based on what data they have
Controller Architectures
Controller Architectures

Spark Controller

- Driver Program
  - RDD Lineage
- Partition Manager Master
  - Generate tasks
- Task Graph
  - T1
  - T2
  - T3
  - T4
- Scheduler
  - Launch tasks
  - Report data placement
- Partition Manager Slave
  - P1
  - W2
  - P2
  - W1
  - P3
  - W2
- Task Queue
  - T1
  - T2
  - T3
  - T4

Starling Controller

- Partition Manager
  - P1
  - W2
  - P2
  - W1
  - P3
  - W2
- Scheduler
  - Decide data placement
  - Broadcast data placement
- Snapshot
  - Specify a stage to snapshot at
- Partition Manager Replica
  - P1
  - W2
  - P2
  - W1
  - P3
  - W2
- Task Queue
  - T1
  - T2
  - T3
  - T4

Spark Worker 1
- Partition Manager Slave
- Task Queue

Spark Worker 2
- Partition Manager Slave
- Task Queue

Canary Worker 1
- Partition Manager Replica
- Task Queue

Canary Worker 2
- Partition Manager Replica
- Task Queue
Partitioning Specification

• Driver program computes valid placements
  – Task accesses determine a set of constraints
    ‣ E.g., A1 and B1 must be on the same node, A2 and B2, etc.

• Micropartitions (overdecomposition)
  – If expected max cores is \( n \), create 2-8n partitions
Managing Migrations

• Controller does not know where workers are in the program
• Workers do not know where each other are in program
• When a data partition moves, need to ensure that destination picks up where source left off
Example Task Graph

LW: local weight
TD: training data
LG: local gradient
GW: global weight
GG: global gradient
Each worker executes the identical program in parallel. Control flow is identical (like GPUs). Tag each partition with last stage that accessed it. Spawn all subsequent stages. Data exchanges implicitly synchronize.
Implicitly Synchronize

Stage gatherG₀ cannot execute until both gradient₁ and gradient₂ complete

Stage again?₀ cannot execute until sum₀ completes
Spawning and Executing Local Tasks

• Metadata for each data partition is the last stage that reads from or writes to it.

• After finishing a task, a worker:
  – Updates metadata.
  – Examines task candidates that operate on data partitions generated by the completed task.
  – Puts a candidate task into a ready queue if all data partitions it operates on are (1) local, and (2) modified by the right tasks.
Two Communication Primitives

• Scatter-gather
  – Similar to MapReduce, Spark GroupBy, etc.
  – Takes one or more datasets as input
  – Produces one or more datasets as output
  – Used for data exchange

• Signal
  – Deliver a boolean result to every node
  – Used for control flow
EVALUATION
Evaluation Questions

• How many tasks/second can Starling schedule?
• Where are the scheduling bottlenecks?
• Does this improve computational performance?
  – Logistic regression
  – K-means, PageRank
  – Lassen
  – PARSEC fluidanimate
• Can Starling adapt HPC applications to changes in available nodes?
A single core can schedule 136,000 tasks/second while using 10% of CPU cycles. Scheduling puts a task on a local queue (no locks); no I/O needed.
Scheduling throughput increases linearly with number of cores. 120,000,000 tasks/second on 1,152 cores (10% CPU overhead)

Bottleneck is data transmission on partition migration.
Workload is 3,600 tasks/second. Queueing delay due to batched scheduling means much higher throughput needed.
Load is 4 tasks/second.
If scheduler can handle 4 tasks/second, queueing delay increases execution time from 1s to 1.75s (75%).
Logistic Regression

Optimal micropartitioning on 32 workers is >18,000 tasks/second

Controller: c4.large instance (1 core)
32 workers: c4.8xlarge instance (18 cores)
Micro partitioning + load balancing runs 3.3x faster than MPI.
PARSEC fluidanimate
(Lagrangian)

Micro partitioning + load balancing runs 2.4x faster than MPI.

Controller: c4.large instance (1 core)
32 workers: c4.8xlarge instance (18 cores)
32 nodes, reprovisioned so computation moves off of 16 nodes onto 16 new nodes. Data transfer takes several seconds at 9Gbps.

Migration times are huge compared to task execution times. Migration not a bottleneck at controller.
Starling Evaluation Summary

• Can schedule 136,000 tasks/s/core
  – 120,000,000/s on 1,152 cores
  – Need to overprovision scheduling/high throughput

• Schedules HPC applications on >1,000 cores

• Micropartitions improve performance
  – Increase required throughput: 5-18,000 tasks/s on 32 nodes

• Central load balancer improves performance
  – 2.4-3.3x for HPC benchmarks

• Can adapt to changing resources
Outline

• HPC background
• Starling scheduling architecture
• Evaluation
• Thoughts and questions
Thoughts

• Many newer cloud computing workloads resembled high performance computing
  – I/O bound workloads are slow
  – CPU bound workloads are fast

• Next generation systems will draw from both
  – Cloud computing: variation, completion time, failure, programming models, system decomposition
  – HPC: scalability, performance
Thank you!

Philip Levis
Hang Qu
Omid Mashayekhi
David Terei
Chinmayee Shah