Starling: A Scheduler Architecture for High Performance Cloud Computing

Hang Qu, Omid Mashayekhi, David Terei, Phil Levis
Platform Lab Retreat
June 2, 2016
High Performance Computing (HPC)

- Use mathematical models to help understand the physical world

- Demand high floating point calculation throughput
  - Take hours, days or weeks to solve a complex model
  - Operate on in-memory data

- A lot of machine learning is HPC
HPC in the Cloud

• HPC has struggled to use cloud resources
• Existing HPC software has certain expectations
  – Cores are statically allocated, never change
  – Cores do not fail
  – Every core has identical performance
• But cloud does not meet those expectations.
  – Elastic resources, dynamically reprovisioned
  – Expect some failures
  – Variable performance
HPC in a Cloud Framework?

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

• 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
  – Queuing 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/second.
  – Scales linearly with number of cores
• HPC benchmarks run 2.4-3.3 faster
Outline

• HPC background
• Starling scheduling architecture
• Evaluation
• Thoughts and questions
HPC Example: Fluid Simulation

• Fluid is modeled as interacting physical variables.
  – Velocity, pressure, external forces, levelset, density, viscosity, marker particles (water interface, splashes, bubbles)...

• Values of each physical variable are sampled over the simulated geometric domain. Each core works on separate domains.
  – Update one physical variable on each domain
  – Synchronize variable values at geometric boundaries
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_levelset(dt);
    // locally calculate, then exchange
    update_particles(dt);
    time += dt;
}

Control flow is explicit in each process.
Fully distributed and no central bottleneck.

Problems in the cloud

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?

HPC code
cloud (EC2, GCE)

HPC code
cloud framework
cloud (EC2, GCE)

Load balancing
Failure recovery
Elastic resources
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 flexible, 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
    • Recorder computation steps if needed (track dependencies)
  – Exchange data
Control Signal

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

Spark Controller

Scheduler

Task Graph

T1

T2

T3

T4

Driver Program

RDD Lineage

Partition Manager

Master

P1

P2

P3

W1

W2

P1

P2

W2

W1

P2

W1

P3

W2

Spark Worker 1

Spark Worker 2

Slaves report local partitions

Launch tasks

Launch tasks

Master answers where a partition is

Generate tasks

Report data placement

Report data placement

Task Queue

Task Queue

Starling Controller

Scheduler

Snapshot

Partition Manager

Replica

P1

P2

P3

W1

W2

P1

P2

P3

W2

W2

W1

W2

P1

P2

P3

W2

P1

P2

P3

W2

Starling Worker 1

Starling Worker 2

Decide data placement

Specify a stage to snapshot at

Broadcast data placement

Broadcast data placement

Scheduler

Task Graph

T4

T2

Task Queue

Task Queue

Driver Program

T1

T3

T1

T3

T2
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, Starling needs 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 the 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 partition it operates on are (1) local and (2) modified by the right tasks.
Outline

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

• Does high scheduling throughput & central control help improve performance?
  – Logistic regression
  – K-means, PageRank
  – Lassen
  – PARSEC fluid animate
Logistic Regression

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

Controller: c4.large instance (1 core)
32 workers: c4.8xlarge instance (18 cores)
Lassen
(unstructured mesh)

Micro partitioning + load balancing runs 3.3x faster than MPI

Controller: c4.large instance (1 core)
32 workers: c4.8xlarge instance (18 cores)
PARSEC fluidanimate (grid)

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