Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud

Omid Mashayekhi, Chinmayee Shah, Hang Qu, Andrew Lim, and Philip Levis
Stanford University

SIGGRAPH 2018
Fluid Simulation Disconnect

- Computing today: the cloud/datacenters
  - Thousands of nodes, commodity parts: failures, "stragglers" common
  - Systems for past 30 years: many cheap parts >> few expensive parts
  - Seminal MapReduce paper: 1600 out of 1800 hosts failed in one run!

- Fluid simulation today: 1990s technologies
  - Single, powerful server (or GPU)
  - Small, high-performance compute cluster
  - HPC: application has complete control over scheduling and placement

- Why? Running software in the cloud is hard
  - Application needs dynamic scheduling, fault tolerance, consistency management
  - Data analytics relies on platforms such as Spark, Hadoop, etc., to handle complexities
  - But we don't want to write brand new simulation libraries for some new framework...
Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud

With a bit of glue code, Nimbus runs an existing, single-threaded simulation library on many machines and hundreds of cores.
Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud

Does this for libraries that have an underlying Eulerian representation, including hybrid methods such as particle level set, fluid implicit particle (FLIP), and affine particle in cell (APIC).
Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud

Run in the cost-effective computing cloud, which has high latency networks, where cores straggle and node failures are common.
How?

• Partition the simulation into many smaller stand-alone simulations across the domain
  - Simulation must have an Eulerian representation

• Between simulation operations, automatically stitch together the sub-simulations
  - Transfer ghost cells after fluid advection
  - Update ghost elements after projection iteration

• Centralized controller monitors, schedules, and controls entire simulation
$152^3$ PhysBAM simulation
24 frames in 335 minutes
1 nodes (1 core)
$256^3$ PhysBAM simulation
24 frames in 268 minutes
8 nodes (64 cores) with Nimbus
Nimbus Programming, Data and System Model
Single-Threaded Simulation Library

velocity

advect(sim.velocity())
parallel {
    apply(advect, {vel, lread_bb}, {vel, lwrite_bb});
    apply(advect, {vel, rread_bb}, {vel, rwrite_bb});
}
Nimbus Parallelizes Simulation

parallel {
    apply(advect, {vel, lread_bb}, {vel, lwrite_bb});
    apply(advect, {vel, rread_bb}, {vel, rwrite_bb});
}

advect(sim.velocity())
advect(sim.velocity())
Two Bits of Glue Code

parallel {
    apply(advect, {vel, lread_bb}, {vel, lwrite_bb});
    apply(advect, {vel, rread_bb}, {vel, rwrite_bb});
}

Nimbus

velocity

adveect(sim.velocity())
adveect(sim.velocity())
Variables are defined over the domain with a partitioning and ghost cell width.

Each variable is divided into a set of disjoint logical objects.

Each logical object has one or more physical instances, stored in memory on nodes.

Application objects are automatically generated and updated from physical objects.
Program Representations

Driver node

Central Controller

Worker nodes
Problem: Data Representations

- Distributed system sees logical and physical objects
- Application library sees application objects, made of many physical objects
- Both expect their objects to be contiguous in memory
Translator Cache

- Maintains consistency between application and physical data objects
- Strawman: maintain both, write-through policy
  - On application write, immediately copy to physical copy
  - Problem: 100% memory overhead, unnecessary copies
- Nimbus approach: write-back cache, free physical objects older than their application objects
  - Instantiate physical object when a transfer starts
  - Application layer is primary data store
  - Physical objects are usually <10% overhead
Problem: Task Rate

- Most centralized controllers scale to 10,000 tasks/second
- Fluid simulations easily execute 100,000 tasks/second
  - Controller quickly becomes a bottleneck at scale: workers fall idle
Controller Cache

• Key observation: iterative simulations have repetitive control traffic/task patterns

• Controller and workers cache blocks of tasks in templates
  - A single control message can instantiate a template and create thousands of tasks
  - A template binds some values (e.g., data objects read and written) but leaves some free (task IDs)
Evaluation
Evaluation Summary

- Nimbus is general, supporting a wide range of Eulerian and hybrid methods
- You can port production quality simulations in ~2,000 lines of code
- You can run bigger, more detailed simulations faster
- System scales past current simulation methods (it's not the bottleneck)
- Nimbus recovers from faults and schedules computations for you
Simulations on Nimbus

- Ported three simulations on two different libraries
  - PhysBAM water and smoke
  - StencilProbe 3D heat diffusion simulation

<table>
<thead>
<tr>
<th>Sim</th>
<th>Library</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat diffusion</td>
<td>118</td>
<td>94</td>
</tr>
<tr>
<td>Water</td>
<td>&lt;1500</td>
<td>620</td>
</tr>
<tr>
<td>Smoke</td>
<td>400</td>
<td></td>
</tr>
</tbody>
</table>
Benefits of Parallelization

152³ PhysBAM simulation
24 frames in 335 minutes
1 nodes (1 core)

256³ PhysBAM simulation
24 frames in 268 minutes
8 nodes (64 cores) with Nimbus
Benefits of Caches

- Heat diffusion application with ghost cell width of 1
- At scale, controller is bottleneck due to high task rate: controller cache allows application to scale
- Translator cache further improves speed by 72%
- Performance caps out at 19.2x: is this Nimbus or the simulation method?
Scalability

- Heat diffusion simulation with caches enabled
- With no data exchanges, Nimbus overhead is <1%
- Data exchanges (simulation method) are the bottleneck
Managing System Complexity

- 256³ PhysBAM water simulation
- 8 nodes (64 cores)
- Fault tolerance/load balancing enabled and disabled
- Nimbus automatically schedules computations away from slow nodes and recovers from node failures
Summary

• Nimbus is an open-source software framework for automatically distributing Eulerian and hybrid simulations on computing cloud nodes
  - Job management, data communication, execution consistency, load balancing, recovery
  - Designed for use with existing simulation libraries

• Nimbus runs higher detail simulations faster

• Key-value stores have dominated distributed systems for 20 years, distributed graphics systems should instead rely on geometry
  - Geometry is concise, defines partitioning as well as locality
  - In Nimbus, geometry is what allows driver program to map to translation cache

• Motivates new distributed graphics simulation methods that scale
Nimbus is a scalable cloud computing framework for computations with sub-millisecond tasks.

Scalability

Nimbus's overhead is negligible even at high scales.

Nimbus has a novel control plane abstraction, called **Execution Templates**, which allows the runtime system handle orders of magnitude higher task rates compared to the best available centralized (Spark) and distributed (Naiad) cloud computing frameworks. At a high level, Execution Templates are dynamically generated execution plans that are installed by the controller at the workers. The key idea behind execution template is that long running applications are iterative in nature (What is an example for iterative patterns within an application?). Instead of scheduling the computations for each iteration from scratch, controller installs the execution plan on each worker once, and instantiates it with new parameters for later iterations. If the load changes or stragglers appear, controller generates new templates to adapt to the dynamic behavior of the cloud, or the data, ensuring no deadline is missed.
Students

Omid  
Chinmayee  
Hang  
Andrew
Scalability

- $1024^3$ PhysBAM water simulation (largest ever?)
- 64 nodes (512 cores)
- 15% slower than hand-tuned PhysBAM MPI libraries
Data Analytics and HPC

- **Spark**
  - Per-task dispatch
  - Centralized, dynamic
  - Explicit data model

- **HPC**
  - Program Flow
  - Distributed, static
  - Implicit data model

** MPI
Data Analytics and HPC

Spark

TensorFlow

Per-task dispatch
Centralized, dynamic
Explicit data model

Data Flow
Distributed, static
Explicit data model

Program Flow
Distributed, static
Implicit data model

MPI
Data Analytics and HPC

- **Spark**
  - Per-task dispatch
  - Centralized, dynamic
  - Explicit data model

- **Legion**
  - Per-task dispatch
  - Distributed, static
  - Explicit data model

- **TensorFlow**
  - Data Flow
  - Distributed, static
  - Explicit data model

- **MPI**
  - Program Flow
  - Distributed, static
  - Implicit data model
Data Analytics Systems

• A centralized controller node schedules and dispatches tasks to worker nodes
  - Tasks load data as needed (try to send tasks to nodes that have data locally)
  - Scheduler can move computations from slow nodes, recover system from failures
  - Much simpler to program than HPC

• Key-value data model, keys hashed to nodes
  - Two types of operators: narrow (read one key) and wide (read all keys)
• My first SIGGRAPH!
  - Most fun ACM conference: I mean, you can buy Magic cards
• VDB: super-interesting and sophisticated data structures
• **But,** every paper and system essentially runs on a single machine
• Large-scale distributed data analytics all the rage in systems research
  - Write a complex program, framework automatically handles all of the hard systems problems, such as data transfer, load balancing, fault recovery
• Pixar's "Brave FX: River Running Through It" talk used Ron's PhysBAM
• Lightbulb: Let's take PhysBAM and run it in the cloud!
What's Hard and Why

- Simulations have much more complex data models and operations
  - Geometric stencils, Eulerian, semi-Lagrangian, projection
  - MAC grids (cell-centered and face-centered), particles, boundary conditions
  - Not just wide or narrow: \((3^d-1)\)-to-1, 1-to-(3^d-1), 1-to-all, all-to-1

- Units of transfer and operation are different
  - Simulations see data in terms of large, contiguous regions (slices/partitions), networked system needs to be able to name and transfer only boundaries (ghost cells)

- Simulations are compute-bound: fast in comparison to data analytics
  - CPU-bound, not I/O bound, so executes orders of magnitude more tasks/second

- Challenge: need a data model that allows the central controller to quickly and correctly manage execution across hundreds/thousands of cores
Geometric Layer

- Variables are defined over the geometric domain with a partitioning and ghost cell width
- Sequential program makes simulation library calls on geometric regions

```
parallel {
    apply(advect, {vel, lread_bb}, {vel, lwrite_bb});
    apply(advect, {vel, rread_bb}, {vel, rwrite_bb});
}
```
Geometric Layer

- Variables are defined over the geometric domain with a partitioning and ghost cell width
- Sequential program makes simulation library calls on geometric regions

```plaintext
parallel {
    apply(advect, {vel, lread_bb}, {vel, lwrite_bb});
    apply(advect, {vel, rread_bb}, {vel, rwrite_bb});
}
```
Logical Layer

- Variables are defined as disjoint geometric partitions (shared memory)
- Every logical object has a read/write order, with a linear write order
- Launcher translates geometric calls into logical calls

```
parallel {
    apply(advect, read:{A, B, C}, write:{A, B});
    apply(advect, read:{B, C, D}, write:{C, D});
}
```
Logical Layer

- Variables are defined as disjoint geometric partitions (shared memory)
- Every logical object has a read/write order, with a linear write order
- Launcher translates geometric calls into logical calls

```
parallel {
    apply(advect, read:{A, B, C}, write:{A, B});
    apply(advect, read:{B, C, D}, write:{C, D});
}
```
Physical Layer

- Variables are memory objects that reside on specific worker nodes
  - Logical object with an associated version number (incremented on each write)
  - One-to-many logical-to-physical mapping

- Central controller transforms logical program into operations on nodes

```plaintext
apply(advect, read:{1, 2, 3}, write:{1, 2});
network_copy(from:2, to:4);
apply(advect, read:{4, 5, 6}, write:{5, 6});
network_copy(from:5, to:3);
```
Physical Layer

• Variables are memory objects that reside on specific worker nodes
  - Logical object with an associated version number (incremented on each write)
  - One-to-many logical-to-physical mapping

• Central controller transforms logical program into operations on nodes

```plaintext
apply(advect, read:{1, 2, 3}, write:{1, 2});
network_copy(from:2, to:4);

apply(advect, read:{4, 5, 6}, write:{5, 6});
network_copy(from:5, to:3);
```
Application Layer

- Variables are the native data objects of the simulation library
- Translator copies between simulation and physical to maintain consistency

```c
advect(sim.velocity);
```

Node 1  |  Node 2

Translator

Application (library)
Program Representations

Driver node

Central Controller

Worker nodes

parallel {
  apply(advect, {vel, lread_bb}, {vel, lwrite_bb});
  apply(advect, {vel, rread_bb}, {vel, rwrite_bb});
}

parallel {
  apply(advect, read:{A, B, C}, write:{A, B});
  apply(advect, read:{B, C, D}, write:{C, D});
}

apply(advect, read:{1, 2, 3}, write:{1, 2});
network_copy(from:2, to:4);
apply(advect, read:{4, 5, 6}, write:{5, 6});
network_copy(from:5, to:3);

advect(velocity); advect(velocity); advect(velocity); advect(velocity); advect(velocity); advect(velocity); advect(velocity); advect(velocity);
Optimized Program Flow

Driver node

Central Controller

Worker nodes

Geometric
Optimized Program Flow

Driver node

Central Controller

Worker nodes
Optimized Program Flow

Driver node

Central Controller

Worker nodes

Geometric

Instantiate template

Instantiate template

Instantiate template

Instantiate template

Instantiate template

Instantiate template

Instantiate template

Instantiate template

Instantiate template