Automatically Distributing and Load Balancing Graphical Fluid Simulations

Chinmayee Shah
Advisor: Prof. Philip Levis

October 23, 2018
Lots of memory
Long compute times
Distributed Simulations are Faster

1 core: > 48 hours
64 cores (8 nodes): 4.5 hours
11x faster
Distributed Simulations Have More Detail

- $152^3$, 1 core: 5.5 hours
- $256^3$, 64 cores, 8 nodes: 4.5 hours
Distributing Applications is Now Cheap

• Earlier, limited access to super-computers and clusters
• Buying, maintaining clusters for simulations expensive
• Cloud provides pay-as-you-use model, possible to rent resources on demand
  - Example: Amazon EC2, Google Cloud, Azure
  - 30 GB, 8 core instance: $0.38/hour
  - Cost for speeding up from more than 2 days to 4.5 hours by distributing over 8 nodes < $14
But almost all graphical fluid simulations run on a single node!
Problem 1: Writing Efficient Distributed Code is Difficult

<table>
<thead>
<tr>
<th></th>
<th>Low Level Frameworks</th>
<th>High Level Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>Manually distribute</td>
<td>Automatically distribute</td>
</tr>
<tr>
<td>Data Flexibility</td>
<td>Flexible data</td>
<td>Impose restrictions</td>
</tr>
<tr>
<td>Existing Codes</td>
<td>Can reuse</td>
<td>Cannot reuse</td>
</tr>
</tbody>
</table>
Problem 2: Difficult to Distribute Work Evenly

• Running efficiently requires distributing work evenly.

• Graphical simulations exhibit spatial and temporal variation in computational load.
Contributions

Thesis: It is possible to automatically distribute and load balance simulations using coarse-grained geometric descriptions about data and tasks.

• A **four layer data model** with Nimbus, a task-based system, to automatically distribute simulations over complex, application-owned data structures.

• **Speculative load balancing** that runs the same simulation over a low resolution grid to predict load and distribute work evenly.
Outline

• Fluid Simulations Background
• Prior Work
• Four Layer Data Model
• Load Balancing
Graphical Fluid Simulations

• Use multiple different data structures
  - particles, grids to store fluid quantities
  - matrices, vectors for linear solves
  - novel techniques over trees, multi-resolution grids

• Spatial computations to evolve fluid

• Distributing requires maintaining shared state across partitions, interleaving computation and communication
Grid and Particle Fields
Store Simulation State

Incompressibility

High resolution details

velocity

mass

velocity

pressure

2 0

4 3
Hybrid: Grids + Particles
Example Simulation

for step : range(10) {
    MoveParticlesInGridVelocity(dt);
    ...
    AdvanceGridVelocity();
    MakeIncompressible();
    ...
}
Distributing: Partition Data, Run Computations over Partitions

Node 1

Node 2

MoveParticlesIn
GridVelocity()

AdvanceGridVelocity()

MakeIncompressible()
Need Data From Neighboring Partitions

Node 1 -> Node 2

Read

Write

MoveParticlesIn
GridVelocity()

AdvanceGrid
Velocity()
Exchange Ghost Data to Satisfy Dependencies

Node 1

Node 2

MoveParticlesIn
GridVelocity()

AdvanceGrid
Velocity()
A 2D partition has $3^2 - 1 = 8$ own + $5^2 - 3^2 = 16$ neighboring ghost regions
A 3D partition has $3^3 - 1 = 26$ own + $5^3 - 3^3 = 98$ neighboring ghost regions
Framework Requirements

• Can automatically distribute simulations
• Easy to write distributed simulations
• Support diverse data structures
• Support novel, optimized data structures
• Support code from existing libraries
Need Support for Diverse Data Structures

for step : range(10) {
    MoveParticlesInGridVelocity(dt);
    ...
    AdvanceGridVelocity();
    MakeIncompressible();
    ...
}
Need Support for Diverse Data Structures

for step : range(10) {
    MoveParticlesInGridVelocity(dt);
    ...
    AdvanceGridVelocity();
    MakeIncompressible();
    ...
}

Reads and Writes Particles, Reads Grid Velocity
Need Support for Diverse Data Structures

for step : range(10) {
    MoveParticlesInGridVelocity(dt);
    ...
    AdvanceGridVelocity();
    MakeIncompressible();
    ...
}

Uses Grid Fields, Sparse Matrices and Vectors
Need Support for Novel Data Structures

Particle Level-Set

Quad/Oct-trees

OpenVDB

Sparse-Paged Grids
Need Support for Existing Production Codes

• PhysBAM:
  - > 100k lines of C++
  - novel simulation techniques

• OpenVDB:
  - optimized data structure
  - many existing codes use it
Outline

• Fluid Simulations Background
• Prior Work
• Four Layer Data Model
• Load Balancing
Legion
Regent
MPI, UPC, GASNet X10
Low Level Systems

Ebb, Simit
Halide, ZPL, Liszt
High Level Systems

App distributes simulation
Distribution
Runtime determines representations

Data Structure Control

App determines representations
Runtime determines representations
MPI, UPC, GASNet, X10

Low Level Systems

App determines representations

Data Structure

Control

Runtime determines representations

App distributes simulation

Distribution

Runtime distributes simulation

High Level Systems

Legion, Regent

Ebb, Simit

Halide, ZPL, Liszt
MPI Example

Node 1

Node 2

MoveParticlesInGridVelocity();
SendParticles();
ReceiveParticles();
AdvanceGridVelocity();
WaitForParticles();
SendGhostVelocity();
ReceiveGhostVelocity();
WaitForGhostVelocity();
MPI: Need to Interleave Communication, Computation

MoveParticlesInGridVelocity();
SendParticles();
ReceiveParticles();
AdvanceGridVelocity();
WaitForParticles();
SendGhostVelocity();
ReceiveGhostVelocity();
WaitForGhostVelocity();
MPI: Need to Implement Data Exchange Calls

```
SendGhostVelocity() {
    for (region : ghost_regions) {
        buffer = SerializeVelocity(region);
        Isend(buffer, region.rank);
    }
}

ReceiveGhostVelocity() {  ...  }

WaitForGhostVelocity() {  ...  }
```
Low Level Systems Require Extensive Effort

Distributing Data and Computations
• Partition and distribute simulation data and computations

Data Exchanges and Synchronization
• Write code to serialize, deserialize, send and receive data
• Wait on unfinished computation and data transfers
Low Level Systems Require Extensive Effort

**Distributing Data and Computations**
- Partition and distribute simulation data and computations

**Data Exchanges and Synchronization**
- Write code to serialize, deserialize, send and receive data
- Wait on unfinished computation and data transfers

**Debugging and Optimizations**
- Perform distributed debugging and performance tuning
- Interleave computation and send/receive calls
- Write code to deal with stragglers and crashes
- Write a load balancer
Higher Level Systems Automatically Distribute Simulations

- Applications often structured as a sequence of computations that read, write a set of variables.
- Systems automatically schedule computations and data exchanges across nodes.
But Higher Level Systems Own Simulation Data Model

• Example data models:
  - graphs in Simit
  - 3D mesh in Liszt
  - tables and uniform grids in Legion and Ebb

• Helps manage system complexities.

• Difficult to express diverse data structures:
  - particles, dynamic sparse grids, multi-resolution grids
  - optimized data structures

• Cannot reuse existing data structure and simulation codes.
Outline

• Fluid Simulations Background
• Prior Work
• Four Layer Data Model
• Load Balancing
Nimbus

Grid and Hybrid Fluid Simulations

Low Level Systems

MPI, UPC, GASNet, X10

Data Structure

Control

Runtime determines representations

App determines representations

App distributes simulation

Distribution

Runtime distributes simulation

High Level Systems

Legion, Regent

Ebb, Simit

Halide, ZPL, Liszt
Different Requirements

- **Simulation Author:**
  Simple driver program using a high level API that is easy to use and hides complexities of distributing

- **Simulation Library Developer:**
  Low level control over simulation data structures
Different Requirements

**Runtime:** To distribute data and computations, the runtime should be able to:

- Analyze and enforce dependencies efficiently
- Replicate and synchronize data across nodes
Four Layer Data Model

1. Geometric:
   - specify partitioning, computations
   - necessary information for distributing

2. Logical:
   - disjoint objects with different accesses
   - track, enforce dependencies efficiently

3. Physical:
   - specific logical objects on different nodes
   - replicate, distribute data

4. Application:
   - complex data objects
   - data in format that simulation expects
1. Geometric Layer

Variable Handle

Write
Read
Write
Read

Application Data
Application Data

Geometric

Driver Program

Launcher

Logical

Controller

Physical

Translator
App
Worker
Geometric Layer: Partitioning and Variables

Domain: $\{(0,0), (6,6)\}$

Partition:

- $x: 2$
- $y: 1$
- Ghost: 1

Create("grid_velocity", partition)
Create("particles", partition)
Geometric Layer:
Tasks over Geometric Partitions

Task: MoveParticlesInGridVelocity
read: 
  { "grid_velocity", outer },
  { "particles", center }
write: 
  { "particles", outer }

Task: AdvanceGridVelocity
read: 
  { "grid_velocity", center },
write: 
  { "grid_velocity", center }
2. Logical Layer

Variable Handle

Write

Read

Write

Application Data

Application Data

Driver (geometric)

Driver Program

Launcher

Logical

Controller

Physical

Translator

App

Application
Need to Refer to Elements with Different Accesses

- MoveParticlesInGridVelocity over partitions read and write the same ghost regions

- **Ghost regions** have a different sequence of task reads and writes from inner regions

- Runtime needs to refer to elements that are accessed differently, for dependency analysis
Runtime Analysis and Operations over Coarse Logical Objects

partition = {
    x : 2,  y : 2,
    ghost : 1
}

Create("grid_velocity", partition)
Create("particles", partition)
3. Physical Layer

Variable Handle

Driver (geometric)

App

Logical

Controller

Translator

Physical

Worker

Application

Launcher

Driver Program

Application Data

Write

Read

Write

Read

Application Data
Physical Objects Identify Instances of Logical Objects

- Multiple instances of logical objects
- Runtime refers to specific instances
Tasks Operate Over Local Physical Data

Node 1

A

B

C

D

Node 2

A

B

C

D

MoveParticles

AdvanceGrid

Velocity

MoveParticles

AdvanceGrid

Velocity
4. Application Layer

Variable Handle

Write

Read

Write

Read

Application Data

Application Data

Driver (geometric)

Driver Program

Launcher

Logical

Controller

Physical

Translator

Application

App

Worker
Application Data: Contiguous Data in Application Format

- **outer**: 25 objects in 2D, 125 in 3D

- Application objects: data over contiguous regions in simulation format
  - multiple variables
  - overlapping regions, partitions
  - arbitrary C++ objects

- Tasks invoke simulation functions over application objects
Managing Application Objects

- Naive: construct application objects for every task
- Doubles memory requirement
- Adds a serialization overhead
Copy Data Required to Satisfy Data Dependencies

- On write:
  - mark physical object as stale
  - copy data out for data transfer

- On read:
  - copy newer data

- 1.7x faster
Nimbus Architecture

- Four layer data model in Nimbus, a task-based system
- Driver program sends partitioning, variable definitions, task launches
- Launcher generates and sends data definitions and task graph to controller
- Controller distributes data, manages replication, schedules tasks
- Workers invoke simulation functions
- Controller provides fault tolerance and straggler mitigation
Evaluation
Distributes Existing Simulations

• 3D heat distribution benchmark
  - Jacobi iterative solver
  - Additional ~200 lines of C++

• PhysBAM water and smoke simulation
  - Particle level-set simulations, used in production
  - More than 40 variables and 8 data structures
  - More than 26 different compute kernels
  - Reuses library, additional ~1.5K lines of C++ serialization
Different Regions: Different Versions, Replicated

Partitioning
\{x : 4, y : 4, z : 4\}

AdvectPhi()
reads : \{“phi”, outer, “face_velocity”, outer\}
writes : \{“phi”, center\}

Iteration 16

Node 1

Node 2

“phi” 2744 logical objects

2,2,2: 3,2,2:
125 logical objects 125 logical objects
share 50, write 9 replicated physical objects

<table>
<thead>
<tr>
<th>Previous:</th>
<th>DataCopy:</th>
<th>AdvectPhi:</th>
</tr>
</thead>
<tbody>
<tr>
<td>v158</td>
<td>v158</td>
<td>v159</td>
</tr>
<tr>
<td>v157</td>
<td>v158</td>
<td>v158</td>
</tr>
<tr>
<td>v158</td>
<td>v158</td>
<td>v159</td>
</tr>
</tbody>
</table>

27 own  98 other  27 own  98 other

<table>
<thead>
<tr>
<th>27 own</th>
<th>98 other</th>
<th>27 own</th>
<th>98 other</th>
</tr>
</thead>
<tbody>
<tr>
<td>v158</td>
<td>v157</td>
<td>v158</td>
<td>v157</td>
</tr>
<tr>
<td>v158</td>
<td>v158</td>
<td>v158</td>
<td>v158</td>
</tr>
<tr>
<td>v159</td>
<td>v158</td>
<td>v159</td>
<td>v158</td>
</tr>
</tbody>
</table>
Simulation Over Application
Data Structures

Partitioning
{x : 4, y : 2, z : 2}

AdvectPhi()
reads : {“phi”, outer, “face_velocity”, outer}
writes : {“phi”, center}

Iteration 16

Task: AdvectPhi()
PhysBAM::WATER_DRIVER<TV> driver(...);
PhysBAM::ScalarArray<T> *phi =
  *GetAppData(“phi”, outer, ...)
PhysBAM::FaceArray<T> grid_velocity =
  *GetAppData(“face_velocity”, center, ...)
driver.Initialize(
  { .phi = phi,
    .grid_velocity = grid_velocity});
driver.AdvectPhi();
PhysBAM: Total Speed-Up

- $256^3$, 30 frames
  - 1 core: > 48 hours
  - 64 cores (8 nodes): 4.5 hours
    - 11x faster

- $256^3$, 70 frames
  - 1 core: > 24 hours
  - 64 cores (8 nodes): ~2 hours
    - 11x faster
Small Overhead Over Hand-Tuned Code

Per Iteration Time (seconds)

PhysBAM Water Simulation
Nimbus Automatically Handles Stragglers and Failures

256³ water simulation over 8 nodes
Outline

• Fluid Simulations Background
• Prior Work
• Four Layer Data Model
• Load Balancing
Load Balancing is Important

• Simulations exhibit spatial and temporal variation in computational load.

• Static, geometric partitioning does not work well for many simulations.

• Need to distribute work evenly, load balance, to run simulations efficiently.
Graphical Fluid Simulations Exhibit Spatial and Temporal Variation

- More computation in regions that contain fluid.
- Computational load varies across space and time.
Most Overloaded Node Determines the Speed

Node 1

Node 2

idle cores

synchronize
Static Geometric Partitioning
Wastes Cores

3.1x
4.2x
2.9x
3.2x
Need Different Assignments for Different Simulations
Need Different Assignments at Different Times

A partition-to-worker assignment that distributes work evenly initially does not necessarily work well for the entire simulation.
Hard to Distribute Work Evenly

- Need to know how fluid evolves
- Automatically determining how fluid evolves requires running the simulation itself!
Low Resolution Simulation Generates Representative Load Estimates

- Low resolution simulation adds negligible overhead.
- Low resolution simulation loses detail.
- But load estimates useful for good load balancing
Low Resolution Simulation Adds Negligible Overhead

Compared to $1024^3$ simulation, a $128^3$ simulation:

- Is $8x8x8 = 512$ smaller, runs 512 times faster
- Needs fewer solver iterations per step
- Runs more than 512 times faster
Speculative Load Balancing
Uses Coarse Load Estimates

step : 21-30

Send Load Estimates → Load Balancer → Reassign Partitions

step : 20
Example Divergence

1024³ simulation

128³ simulation
Synchronize if Diverge

step: 21-30

Send Load Estimates

Load Balancer

Reassign Partitions

Divergence

Reset

Sampled Particles

step: 20
Distributing Partitions Given Load Estimates

• Need to distribute work evenly across workers.
  - Existing technique, micro-partitioning to distribute work finely
  - Minimize maximum load (load on most overloaded node)

• Need to keep the overhead of load balancing low.
  - Reduce frequency of load balancing by computing a single assignment for several steps at a time
  - Add a migration cost to further reduce number of partitions migrated during reassignment
Micro-Partition to Distribute Work Evenly
Micro-Partition to Distribute Work Evenly
Micro-Partition to Distribute Work Evenly
Micro-Partition to Distribute Work Evenly

Node 1

Node 2
Since time proportional to load on most overloaded node, want to minimize maximum load.

Computing exact solution is expensive (NP-hard).

Approximate solution using a fast, greedy algorithm.

A random assignment
Greedy Algorithm

Assign the next largest partition to least loaded node.
Assignment changes. Moving partitions is expensive.
Greedy Algorithm Over Time

- **Reducing migrations**: Compute a single assignment that works well for several future steps.
- **Modified algorithm for series**: Assign partition with next largest mean load, to node with least total cost for the next.
- **Cost**: Sum of load on most overloaded node at each time, plus migration cost if new assignment moves partition.
Greedy Algorithm Over Time

Cost at each step = load on most overloaded node,
Total cost = sum of cost at each step + migration cost
Greedy Algorithm Over Time

Partition a

<table>
<thead>
<tr>
<th>t = 1</th>
<th>a1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 2</td>
<td>a2</td>
</tr>
<tr>
<td>t = 3</td>
<td>a3</td>
</tr>
</tbody>
</table>

Partition b

<table>
<thead>
<tr>
<th>t = 1</th>
<th>b1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 2</td>
<td>b2</td>
</tr>
<tr>
<td>t = 3</td>
<td>b3</td>
</tr>
</tbody>
</table>

Partition c

<table>
<thead>
<tr>
<th>t = 1</th>
<th>c1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 2</td>
<td>c2</td>
</tr>
<tr>
<td>t = 3</td>
<td>c3</td>
</tr>
</tbody>
</table>

Partition d

<table>
<thead>
<tr>
<th>t = 1</th>
<th>d1</th>
</tr>
</thead>
<tbody>
<tr>
<td>t = 2</td>
<td>d2</td>
</tr>
<tr>
<td>t = 3</td>
<td>d3</td>
</tr>
</tbody>
</table>

Node 1

- c1
- c2
- c3

Node 2

- a1
- a2
- a3
- d1
- d2
- d3
Greedy Algorithm Over Time

Partition a
- t = 1: a1
- t = 2: a2
- t = 3: a3

Partition b
- t = 1: b1
- t = 2: b2
- t = 3: b3

Partition c
- t = 1: c1
- t = 2: c2
- t = 3: c3

Partition d
- t = 1: d1
- t = 2: d2
- t = 3: d3

Node 1
- c1: b1
- c2: b2
- c3: b

Node 2
- a1: d1
- a2: d2
- a3: d3
Evaluation
Experiment Details

• Basic FLIP simulation, uses sparse OpenVDB voxels to store grid fields
  - no advanced effects
  - 8 particles per cell
  - Conjugate gradient method, with block incomplete diagonal Cholesky preconditioner

• High resolution simulation over $1024^3$ grid, low resolution simulation over $128^3$ grid
Experiment Details

• Results for 4 simulation configurations

• Total speedup when distributed over 8 nodes (8 cores each) over 1 node (8 cores) for:
  - static, geometric partitioning
  - reactive load balancing, using current load
  - speculative load balancing, using future load

• Reactive and speculative methods load balance every 30 steps
Speed-Up With Changing Load

Sphere Drop
One Way Dam
Two Way Dam
Sources

Frames 1-100
Frames 1-100
Frames 1-150
Frames 1-150

1  3  5.3  6.9  29%
1  1  4.1  6.2  16%
1  1  3  5.4  17%
1  1  2.7  4.8  15%

Speed-Up Graphs for different load scenarios.
Speed-Up With Changing Load

Total overhead from low resolution simulation

< 0.5%
Speed-Up With Stable Load

Sphere Drop
One Way Dam
Two Way Dam
Sources

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
10%
10%
10%
10%

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
5.9
5.4
3.4
1

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
7.8
7.3
4.4
1

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
5.7
5.4
2.9
1

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
6
6
3.3
1

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
8
8
5
1

Frames 101-200
Frames 101-200
Frames 151-300
Frames 151-300

Single
Geom.
React.
Spec.
0
0
0
0
Summary

• It is challenging to distribute simulations efficiently:
  - Distributing simulations over complex data structures, and from existing libraries is difficult.
  - Variation in computational load makes distributing work evenly, and running simulations efficiently, challenging.

• Contributions:
  - A four layer data model that separates different requirements. Using this, Nimbus can distribute complex fluid simulations.
  - Speculative load balancing, that uses load estimates from a low resolution simulation, to load balance simulations.
Publications


• Omid Mashayekhi, Chinmayee Shah, Hang Qu, Andrew Lim, Philip Levis. *Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud*. Published in ACM Transactions on Graphics (TOG), 2018.


Thank You