ACM SIGGRAPH / Eurographics Symposium on Computer Animation 2018
T. Beeler and N. Thuerey
(Guest Editors)

Volume 37 (2018), Number 6

Distributing and Load Balancing Sparse Fluid Simulations

C. Shah, D. Hyde, H. Qu, and P. Levis

Stanford University, USA

Abstract

This paper describes a general algorithm and a system for load balancing sparse fluid simulations. Automatically distributing sparse fluid simulations efficiently is challenging because the computational load varies across the simulation domain and time. A key challenge with load balancing is that optimal decision making requires knowing the fluid distribution across partitions for future time steps, but computing this state for an arbitrary simulation requires running the simulation itself. The key insight of this paper is that it is possible to predict future load by running a speculative low resolution simulation in parallel. We mathematically formulate the problem of load balancing over multiple time steps and present a polynomial time algorithm to compute an approximate solution to it. Our experimental results show that distributing and speculatively load balancing sparse FLIP simulations over 8 nodes speeds them up by 5.3× to 7.9×, and that speculative load balancing generates assignments that perform within 20% of optimal.

CCS Concepts

•***Computing methodologies*** → ***Distributed computing methodologies;*** *Distributed simulation; Computer graphics;*

1. Introduction

Modern computer graphics rely on fluid simulations to create special effects such as floods, gushing rivers, smoke, fire, and stormy seas. Generating these effects at high resolution is both memory and compute intensive. Recent data structures such as OpenVDB [MLJ*13] and sparse paged grids [SABS14] use dynamic sparse representations such as trees to enable larger and more efficient single node simulations. By only storing simulation state where fluid is present, these data structures enable effects like trails of smoke and rivers over enormous volumes.

While sparse data structures are more memory efficient, simulations using them are still constrained by the computational performance of a single node. Simulations distributed across many machines and hundreds of cores run faster and have greater detail [MSQ*18]. Automatically distributing dynamic sparse grid simulations is challenging — each simulation step is limited by the speed of the slowest node. A poor partitioning can result in a majority of the nodes falling idle, waiting on the slowest, most overloaded node. Furthermore, the topology of a sparse grid changes as fluid moves through the domain, requiring dynamically managed partitions and neighbors. As a result, simple partitioning strategies such as geometrically dividing the domain across nodes perform poorly and waste cores.

Distributing partitions such that work is evenly spread across nodes requires knowledge of the application. Historically, simulation developers were expected to know how the simulation evolves and specify a partitioning and assignment to the launcher. For instance, MPI [Sni98], a message passing interface that is widely used for writing multi-node simulations, uses a *hostfile* to map process ranks to nodes at launch time. This approach has two drawbacks. First, such a manual assignment is error prone. Second, there are simulations such as a dam break where the distribution of fluid across most of the domain changes with time. It is necessary to dynamically reassign partitions to keep these simulations running efficiently.

Optimally deciding how to assign partitions to nodes requires knowledge about how fluid distribution will evolve in the future. Doing this automatically for any arbitrary simulation requires running the simulation itself. Existing HPC approaches heuristically approximate this using the current load distribution across partitions instead [KK93, PGDS*14]. Such a *reactive* approach produces assignments that are sub-optimal for future time steps, as the computed assignments do not account for temporal variation. Since graphical simulations often use implicit methods with large time steps, with some techniques enabling very large CFL numbers [LCPF12], the computational load across partitions can change rapidly. Effectively load balancing using just the current information requires frequent load balancing. This is inefficient because the overhead for load balancing scales with the size of the simulation and the number of partitions and nodes; each load balancing step requires exchange of control information and serializing and transferring simulation data between nodes.

The key insight of this paper is that a system can quickly run a low resolution simulation to estimate future load distribution because of how rapidly simulations scale in size and time. It proposes a new approach, *speculative* load balancing, to optimize partition-to-worker assignments for temporal variation. Speculative load balancing predicts future computational load across partitions by running a low resolution simulation in parallel and uses this information to automatically compute assignments that perform well in practice. The small execution time for the low resolution simulation makes it possible to do this with negligible overhead. This paper makes the following contributions:

1. A design and techniques for distributing and load balancing simulations over sparse data structures such as OpenVDB.
2. An algorithm to compute partition-to-worker assignments using load estimates for future time steps from a low resolution simulation, that takes polynomial time in the number of partitions.
3. A system architecture for speculatively load balancing using a parallel low resolution simulation.
4. Experimental evaluations that show performance benefits of distributing and speculatively load balancing sparse fluid simulations. Distributing and speculatively load balancing over 8 nodes speeds up simulations by 5.3× to 7.9×. Speculative balancing performs within 20% of the ideal case when work is distributed evenly across all nodes for every time step.

The source code and libraries for this system are open source and freely available for use at `https://sing.stanford.edu/nimbus/speculative-lb.tar.gz`.

2. Related Work

2.1. Simulation Techniques, Data Structures and Libraries

Graphical simulations use techniques ranging from grid or particle based approaches to hybrid methods, meshes, and adaptive structures. SPH [GM77, DC*96] models use a purely particle based, Lagrangian approach. Grid-based methods typically use a staggered MAC grid [HW65] for greater accuracy, an unconditionally stable semi-Lagrangian advection scheme for advecting quantities [Sta99], and a pressure Poisson solver for enforcing incompressibility [FSJ01]. Particle-in-cell based approaches such as FLIP, PIC and APIC [ZB05, Har62, JSS*15] advect particles to reduce dissipation, and use grids to perform computations, such as enforcing incompressibility, that are difficult over particles. The particle-level set method [EFFM02] uses a thin band of particles near the interface to improve visual accuracy. Narrowband FLIP maintains particles in a narrow band near the surface and advects quantities directly over the grid in the interior [FAW*16]. Other approaches use meshes to track fluid surfaces [WMFB11].

OpenVDB [MLJ*13] and sparse paged grids [SABS14] allow efficient representation of sparse quantities over large domains. Adaptive data structures such as octrees [LGF04, AGL*17] and chimera grids [EQYF13] use more detailed grids around visually interesting regions such as surfaces and vortices. Other approaches use a high resolution sparse grid to capture the surface more accurately, while enforcing incompressibility over a lower resolution grid [GBW16]. All these simulation methods show significant variation in computational requirements over time and space, and can benefit from load balancing. Load balancing simulations becomes even more important with recently developed techniques that allow very large time steps [LCPF12].

PhysBAM [DHF*11] is an open-source library that supports fluid simulations using the particle-levelset method, and statically distributes simulations over multiple cores and nodes using MPI [Sni98]. Mantaflow [TP16], another library, supports FLIP and narrowband FLIP, and OpenVDB volumes. Mantaflow supports parallelism over a single node using OpenMP [DM98], Intel Threading Building Blocks [Phe08] and CUDA. Commercial visual effects software such as Maya [May18], Houdini [Hou18] and Blender [Ble18] that support fluid simulations also use static partitioning. Speculative load balancing can benefit all of these, as fluid simulations show variation in fluid distribution and computational load across space and time.

Low resolution simulations generated using simulation or animated by artists have been successfully used for guiding high resolution simulations [Chr10, NCZ*09, NB11, RTWT12]. This paper uses low resolution simulation to speculatively compute partition-to-worker assignments for a high resolution simulation.

2.2. Load Balancing

Many partitioning algorithms have been developed to distribute partitions evenly and minimize communication across nodes. Graph partitioning algorithms can compute an effective assignment for partitions with arbitrary topology given an estimate for work for each partition and for communication overhead between partitions [KK96, CBD*07, Kar03]. Geometric partitioning algorithms such as recursive bisection [BB87] and space-filling curves [PB94] can be used to map partitions for simulations over grids and particles. Greedy list scheduling with partitions sorted by load produces an assignment with theoretical guarantees on imbalance [KT06]. However, a naive implementation of greedy load balancing can produce a large communication overhead from ghost exchanges.

As a simulation evolves and the work across partitions changes, it is necessary to remap partitions to keep the simulation running efficiently. Work-stealing approaches redistribute load across cores by having idle cores fetch work from busy cores on the same node [FLR98, Phe08] or remote nodes [LKK14]. Work-stealing approaches work well on shared memory machines, but can suffer from large overhead due to repeated rebalancing in a multi-node setting. Many dynamic load balancing schemes use an estimate of load across partitions from the current step to recompute partition-to-worker assignment and rebalance load for future steps [SKK00, PGDS*14, KK93, BDF*07]. This estimate is either provided by the application — using information such as number of particles or cells — or computed by the runtime using the total compute time over each partition. Scratch-Remap algorithms recompute a new assignment from scratch and then map groups to worker nodes using similarity with the previous assignment, so as to minimize migration costs [OB98]. Cut-and-paste methods attempt to reduce the migration costs by incrementally moving partitions from overloaded nodes to underloaded nodes, but can impose large communication overhead as many neighboring partitions get assigned to different nodes [SKK00]. Diffusive load balancing schemes model the migration problem as a flow problem and incrementally update assignments by moving partitions from overweight nodes to neighboring nodes, so as to minimize a combination of migration overhead and edge cut [OR94, HBE98].

While traditional load balancing methods use a single scalar to represent load for each partition, multi-dimensional load balancing problems generalize this to multi-dimensional vector loads, possibly representing different quantities such as CPU, memory, and network requirements. Recent work on multi-dimensional load balancing includes techniques to minimize the maximum load across each dimension, and across all workers [BOVVDZ16, CK04].

2.3. Systems and Techniques for Distributing Simulations

MPI [Sni98] is a parallel runtime that provides various communication primitives for application writers to send and receive messages between processes. Computation and communication are interleaved, so writing correct and high performance code requires extensive developer effort and careful placement of data exchanges. Charm++ [KK93] resembles MPI, but includes additional support to load balance applications using runtime measurements.

Task-based systems such as Canary [QMSL18], Legion [BTSA12], HPX [KHAL*14] , Uintah [HMB12] and Nimbus [MQSL17, MSQ*18] model computations as a sequence of tasks that read or modify data objects. These systems automatically schedule computations such that inter-task dependencies are satisfied and overlap computation with communication. They also include support for remapping partitions. Unlike other systems that use a centralized controller, Canary uses an asynchronous control plane that can scale out to a large number of nodes and high task rates. Micro-partitioning or over-decomposition reduces communication overhead from ghost data exchanges between neighboring partitions. It does so by partitioning the simulation domain so that there are multiple partitions per core, and by scheduling computations and communication so that they overlap [BVK09]. We use Canary to distribute our simulations, and micro-partitioning to mask the delay of ghost data exchanges.

Domain specific languages such as Liszt [DJP*11], Ebb [BSL*16], Simit [KKRK*16], and Regent [SLT*15] automatically parallelize simulations, but do not automatically load-balance simulations.

3. Distributing and Load Balancing

Distributing a simulation has two benefits. First, it enables larger simulations that can generate higher amounts of visual detail [MSQ*18]. Second, distribution makes the same simulation complete faster, reducing turnaround times from several days to less than a day. Since the amount of time required to compute over a partition is a function of the amount of fluid and details such as the fluid-air interface and vortices it contains, computation time varies over different partitions. In order to achieve speedup proportional to the number of nodes, it is important to assign partitions such that work is evenly distributed across these nodes. For instance, using twice the number of cores for a dam break simulation should reduce the simulation completion time by half, but if half of the cores sit idle, the simulation will still take the same amount of time.

3.1. Distributing Sparse Simulations

We micro-partition the simulation domain so that there are 4 to 16 partitions per core. Micro-partitioning is useful for two reasons. First, it enables load balancing. With a single partition per core, the time that a simulation step takes is determined by the core running the partition with the most fluid. With multiple partitions per core, it becomes possible to distribute work more evenly by assigning each node its fair share of partitions with high computational load. Second, micro-partitioning helps mask the overhead of ghost data exchanges with computation [BVK09]. A core running computations over a single partition cannot start the computation until all data dependencies are met. When these dependencies include ghost data from neighboring partitions, the core blocks while waiting for all data. With multiple partitions per core, it can instead start computing over another partition with dependencies satisfied.

We use OpenVDB [MLJ*13] for representing sparse fluid data. OpenVDB is a volumetric data structure for representing large, sparse grids. It uses dynamic B+ trees with large branching factors, fast bit operations, and caching of nodes to provide memory efficient sparse grids with fast sequential and stencil access to voxel data. We split a large OpenVDB simulation into smaller simulations by geometrically partitioning OpenVDB voxels into regions. Each partition is a single-threaded sub-simulation: it maintains a separate OpenVDB tree with its own copy of the root node, internal nodes, leaf nodes, and voxel data for the region it owns and for neighboring ghost (shared) regions. The system automatically schedules and stitches together the sub-simulations to produce a single large simulation, similar to Nimbus [MSQ*18].

3.2. Speculative Load Balancing

A common way to distribute simulations is to statically divide the simulation domain geometrically across nodes. Using the distribution of fluid across partitions at the beginning and knowledge about how the distribution evolves over time, a developer can manually specify a partitioning and assignment that distributes work somewhat evenly for most time steps. However, there are simulations, such as a dam break, where the amount of fluid in each partition varies greatly over time, and for which no static assignment of partitions works well. Dynamically reassigning partitions to balance load can lead to speedups of 1.7-2.7$\times$ over static assignment.

Existing HPC load balancing approaches reactively balance load by using current partition load estimates to periodically recompute assignments and redistribute work [SKK00, PGDS*14, KK93, BDF*07]. Greedily optimizing for the current distribution without accounting for temporal variation, however, can produce assignments that perform sub-optimally for later time steps. If multiple partitions with increasing load are assigned to the same worker, the worker will slow down progress for later time steps. With graphical simulation techniques that allow large time steps, it becomes even more important to account for variation as the amount of fluid in each partition changes more rapidly with time.

Accounting for load variation over time requires information about the future. In general, it is not possible to do this automatically for an arbitrary simulation without running the simulation itself. The key idea that speculative load balancing uses is that it is possible to quickly estimate fluid distribution over time for graphical fluid simulations over sparse uniform grids by running a low resolution simulation in parallel. Since the amount of time required to run a simulation scales super-linearly with the simulation size, the overhead from running a low resolution simulation is very low. For instance, a 3D simulation that is 8 times smaller in each dimension can advance a single step $8^3 = 512$ times faster. Furthermore, the low resolution simulation requires fewer time steps to advance the same frame time and fewer solver iterations to enforce incompressibility. With a small execution time that is further masked by running in parallel with the high resolution simulation, the low resolution simulation presents negligible overhead.

Speculative load balancing estimates computational load over time by running a low resolution simulation ahead of the high resolution simulation. It uses these estimates to periodically recompute an assignment of partitions that is close to optimal for the entire duration of the assignment. The remainder of this section mathematically formulates the problem of load balancing over multiple time steps and develops a polynomial time algorithm to compute an approximate solution to the problem.

3.3. Model and Cost Function

Let N be the total number of worker nodes over which the high resolution simulation is distributed and P be the total number of partitions. Let a T-dimensional vector $\boldsymbol{B}_p$ denote the computational load for partition p for T time steps, and $\boldsymbol{B}_p^t$ denote the load for p at time step t. Computational load for a partition for a step is a measure of the amount of time it takes to perform computations and advance the step over the partition. For a FLIP simulation with a uniform number of particles per fluid cell, this is proportional to the number of fluid cells the partition contains.

The ideal distribution of load across all nodes is when the computational load is distributed evenly across all nodes, so that each node has $\hat{C}^t = \sum_{p=1}^{P} \boldsymbol{B}_p^t / N$ load at every time step. Let $\boldsymbol{A}$ denote the partition assignment matrix so that $\boldsymbol{A}_p^i = 1$ indicates that partition p is assigned to node i. Since the time that step t takes is proportional to the maximum amount of load a worker has at step t, we define the cost C^t of an assignment at time t as:

$$C^t = \max_{i=1}^{N} \sum_{p=1}^{P} \boldsymbol{A}_p^i \boldsymbol{B}_p^t \tag{1}$$

The total cost of assignment $\boldsymbol{A}$ for T time steps is the sum of cost for each step:

$$C = \sum_{t=1}^{T} \max_{i=1}^{N} \sum_{p=1}^{P} \boldsymbol{A}_p^i \boldsymbol{B}_p^t \tag{2}$$

The above cost estimate ignores inter-partition communication cost. However, since we use micro-partitioning as described in Section 4, computations and ghost data exchanges overlap, reducing the overhead from ghost data exchanges [BVK09].

The load balancing problem for T time steps is now as an integer program, where the optimal assignment $\boldsymbol{A}$ is the solution to the following optimization problem:

$$\text{minimize} \sum_{t=1}^{T} \max_{i=1}^{N} \sum_{p=1}^{P} \boldsymbol{A}_p^i \boldsymbol{B}_p^t$$

Algorithm 1 Greedy List Scheduling For A Single Step

```
 1: $P \leftarrow$ number of partitions
 2: $\boldsymbol{B} \leftarrow$ $P$-dimensional list of scalar load for each partition
 3: $N \leftarrow$ number of worker nodes
 4: procedure GREEDYLISTSCHEDULE($P$, $\boldsymbol{B}$, $N$)
 5:     $\boldsymbol{L} \leftarrow$ $N$-dimensional worker load list, initialize to zero
 6:     $\boldsymbol{A} \leftarrow$ $P$-dimensional partition assignment list
 7:     $\boldsymbol{I} \leftarrow$ IndicesSortedInDescendingOrderBy($\boldsymbol{B}$)
 8:     for $p$ in $\boldsymbol{I}$ do
 9:         $n \leftarrow$ ArgMin($\boldsymbol{L}$)
10:         $\boldsymbol{L}[n] \leftarrow \boldsymbol{L}[n] + \boldsymbol{B}[p]$
11:         $\boldsymbol{A}[p] = n$
12:     return $A$
```

subject to:

$$\begin{aligned} &\boldsymbol{A}_p^i \in \{0,1\} \ \forall p,i \\ &\sum_{i=1}^{N} \boldsymbol{A}_p^i = 1 \ \forall p \end{aligned} \tag{3}$$

The constraints $\boldsymbol{A}_p^i \in \{0,1\}$ and $\sum_{i=1}^{N} \boldsymbol{A}_p^i = 1 \ \forall p$ ensure that each partition is assigned to exactly one node.

3.4. Optimizing for a Single Step

For a single time step, the integer program in Equation 3 reduces to minimizing $C^t = \max_{i=1}^{N} \sum_p \boldsymbol{A}_p^i \boldsymbol{B}_p^t$. This problem is NP hard [KT06]. Simplifying the integer program to a linear program so that $\boldsymbol{A}_p^i$ is constrained to be in $[0,1]$ instead of $\{0,1\}$ does not produce a useful assignment. Since the problem is symmetric in p and i, the optimal solution to the relaxed problem is $\boldsymbol{A}_p^i = \frac{1}{N}$, that is, to split each partition equally across all nodes.

We propose using a polynomial time algorithm, greedy list scheduling, to compute an approximate solution to the single step load balancing problem [KT06]. Greedy list scheduling with tasks sorted by load generates an assignment that is guaranteed to be within $\frac{4}{3}$ of the optimal [Gra69]. It visits tasks in decreasing order of load and incrementally assigns each task to the worker node with the least amount of total work assigned so far. Algorithm 1 describes the algorithm in detail.

3.5. Optimizing for Multiple Steps

This section generalizes greedy list scheduling for a single time step to multiple time steps. Greedy list scheduling takes in a scalar load estimate for each partition to sort the partitions and determine the worker with the least amount of work. A naive way to extend this approach to multiple time steps is to compute a scalar representing the mean load $\hat{\boldsymbol{B}}_p$ for each partition,

$$\hat{\boldsymbol{B}}_p = \frac{1}{T} \sum_{t=1}^{T} \boldsymbol{B}_p^t \tag{4}$$

and run greedy list scheduling using the mean partition load, so as to minimize $\max_{i=1}^{N} \sum_{p=1}^{P} \boldsymbol{A}_p^i \hat{\boldsymbol{B}}_p^t$. The problem with this approach

Algorithm 2 Scheduling For Multiple Time Steps

```
1: $P \leftarrow$ number of partitions
2: $T \leftarrow$ number of time steps
3: $\boldsymbol{B} \leftarrow P \times T$ partition load matrix
4: $N \leftarrow$ number of worker nodes
5: procedure MULTISTEPSCHEDULE($P$, $T$, $\boldsymbol{B}$, $N$)
6:     $\boldsymbol{L} \leftarrow N \times T$ worker load matrix, initialize to zero
7:     $\boldsymbol{A} \leftarrow P$-dimensional partition assignment list
8:     $\hat{\boldsymbol{B}} \leftarrow \frac{1}{T} \times \sum_{t=1}^{T} \boldsymbol{B}[:,t]$
9:     $\boldsymbol{I} \leftarrow$ IndicesSortedInDescendingOrderBy($\hat{\boldsymbol{B}}$)
10:    for $p$ in $\boldsymbol{I}$ do
11:        $\boldsymbol{L}' \leftarrow N \times T$ matrix, initialized to zero
12:        for $i$ in range($N$) do    ▷ Load on $i$ if $p$ is assigned to it
13:            $\boldsymbol{L}'[i,:] \leftarrow \boldsymbol{L}[i,:] + \boldsymbol{B}[p,:]$
14:        $\boldsymbol{D}_p \leftarrow N$-dimensional cost list, initialize to zero
15:        for $t$ in range($T$) do    ▷ Cost of assigning $p$ to node $i$
16:            $\boldsymbol{D}_p[i] \leftarrow \boldsymbol{D}_p[i] + \max(\boldsymbol{L}'[i,t], \boldsymbol{L}[-i,t])$
17:        $n \leftarrow$ ArgMin($\boldsymbol{D}_p$)
18:        $\boldsymbol{L}[n,:] \leftarrow \boldsymbol{L}[n,:] + \boldsymbol{B}[p,:]$
19:        $\boldsymbol{A}[p] = n$
20:    return $A$
```

is that multiple partitions with similar trends, increasing or decreasing load, may get assigned to the same node. When the temporal variation is large, this results in a poor assignment.

The key idea that makes greedy list scheduling work well for a single step is that a sub-optimal assignment for tasks with a small load is better than a sub-optimal assignment for tasks with a large load. By visiting partitions in the decreasing order of their load, greedy list scheduling assigns and spreads out the bigger partitions first. We generalize this approach for computing assignments with T-dimensional loads. Since there is no clear way to order partitions with T-dimensional loads, we sort them by their mean load, $\hat{\boldsymbol{B}}_p$ given by Equation 4. Ordering partitions by their mean load works well in practice, as partitions with large loads get assigned first.

Algorithm 2 describes our algorithm. The algorithm visits each partition in decreasing order of mean load and computes the incremental cost of assigning the partition to each node. The incremental cost $\boldsymbol{D}_p^i$ of assigning a partition p to a node i is:

$$\boldsymbol{D}_p^i = \sum_{t=1}^{T} \max_{j=1}^{N}(\text{load at node } j \text{ assuming } p \text{ is assigned to } i)$$

The algorithm assigns the partition p to the node with the least amount of cost $\boldsymbol{D}_p^i$. As the incremental cost for assigning a partition with increasing load to a node with increasing load is higher than the cost for assigning the partition to a node with decreasing load when the two nodes have the same mean load, this algorithm outperforms the naive extension that runs greedy list scheduling with per partition mean load.

Speculative load balancing uses this algorithm to compute an assignment using partition load estimates for multiple time steps from a low resolution simulation. Section 5 presents results for speculative load balancing, demonstrating that this algorithm produces assignments that perform within 20% of the ideal case.

Algorithm 3 Basic FLIP/PIC Simulation Algorithm

```
1: Initialize particle positions and velocities
2: for each frame do
3:     while frame not done do
4:         Compute time step
5:         Transfer particle velocities to grid
6:         Save grid velocities
7:         Add forces
8:         Apply boundary conditions
9:         Make velocity divergence-free
10:        Compute velocity update over grid
11:        Update particle velocities by interpolating grid values
12:        Update particle positions using updated velocities
13:    Save particle positions and velocities
```

4. System Design and Implementation

We use Canary [QMSL18] for distributing a basic FLIP/PIC simulation that uses OpenVDB grids for storing simulation quantities. Canary uses an asynchronous control plane to provide the scheduling flexibility of a centralized controller with the scalability of a dataflow system such as MPI.

4.1. FLIP Simulation

We implemented a basic FLIP/PIC algorithm [ZB05] for simulating water in C++. Algorithm 3 lists all steps for reference. Our simulation uses 8 particles per cell, and a block diagonal preconditioner using incomplete Cholesky factorization. Our solver does not account for advanced effects such as surface tension and viscosity [**?**, **?**]. We expect that our methodology would provide similar benefits under variations of the basic FLIP algorithm.

Our simulation uses a staggered grid for velocity and stores grid fields such as velocity and pressure using separate OpenVDB grids. We also use OpenVDB for representing particles as a grid of linked lists of buckets of particles. Only those voxels that contain fluid are active, the rest default to the background value. The grids use trees with one level of internal nodes and a branching factor of 2^3 along each dimension, for internal and leaf nodes.

4.2. Distributing With Canary

An application developer provides a driver program, and serialization/deserialization methods for the simulation variables. A driver program specifies three things: (1) typed simulation variables that are either scalars that represent global variables such as the time step, or fields such as OpenVDB grids, (2) a sequence of Canary tasks over typed simulation variables with read and write data dependencies for each task, and (3) a fixed partitioning to use for the simulation variables.

A task may be a compute task that reads and writes OpenVDB

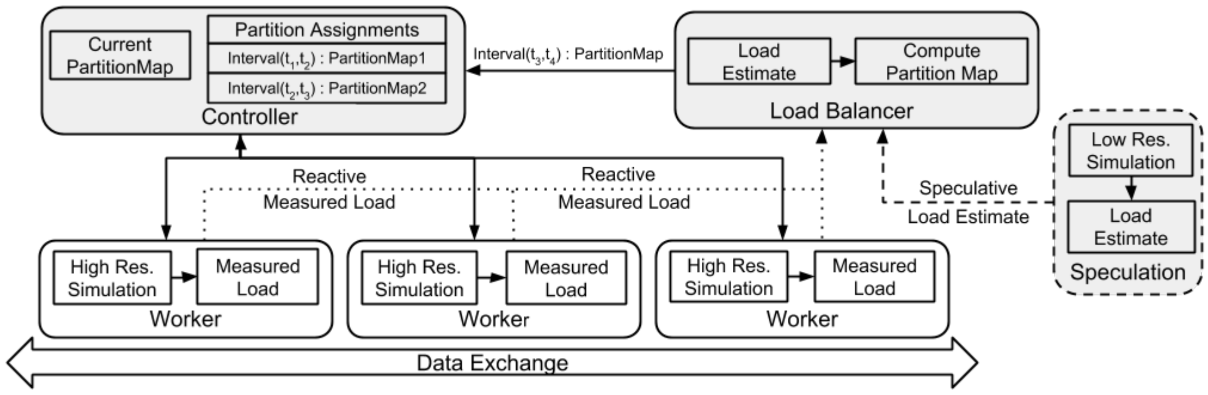
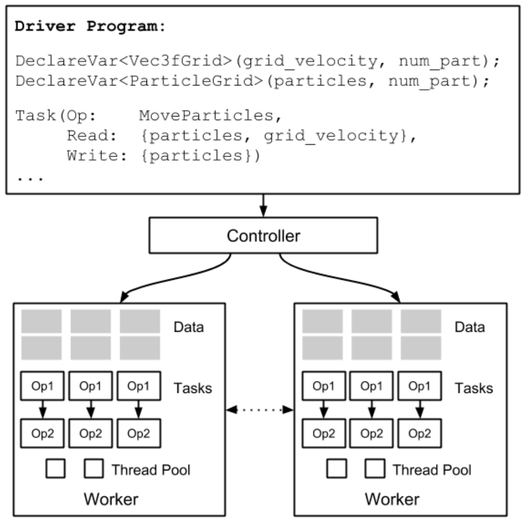


Figure 1: *Speculative load balancing uses load estimates for future steps from a parallel low resolution simulation. Reactive load balancing computes assignments based on current load. A central controller keeps a list of contiguous simulation time intervals and assignments to use for each interval, as computed by the load balancer, and issues required migrations whenever there is an update to partition map.*



```
Driver Program:

DeclareVar<Vec3fGrid>(grid_velocity, num_part);
DeclareVar<ParticleGrid>(particles, num_part);

Task(Op:    MoveParticles,
     Read:  {particles, grid_velocity},
     Write: {particles})
...
```

Figure 2: *A driver program declares typed variables and launches a sequence of tasks over these variables. The system automatically assigns partitions schedules sub-simulations over partitions.*

data and invokes single-threaded simulation functions such as moving particles or transferring particle quantities to grid, a scatter task that outputs data to be sent to other partitions, or a gather task that reads data received from other partitions. Canary transforms the driver program into a series of parallel tasks over partitions, assigns partitions and tasks to worker nodes [QMSL18], and manages data exchanges between workers. Each worker node uses a thread pool to execute tasks over all the partitions it owns. Tasks over empty partitions with no active voxels return immediately. The system automatically schedules sub-simulations over partitions to produce a single large simulation. Figure 2 illustrates this.

We added functions to serialize and deserialize OpenVDB grid data over a given geometric region. These are invoked for ghost data exchanges and data migration.

4.3. Load Balancing

A central load balancer uses load estimates to periodically compute a partition map specifying a worker for each partition, and sends the computed map, along with the simulation time interval over which to use it, to the controller. Reactive load balancing uses the current load directly from the high resolution simulation and Algorithm 1; speculative load balancing uses load estimates for future steps from the low resolution simulation and Algorithm 2.

We use the number of fluid cells in a partition as an estimate of the computational load for the partition. As Figure 1 shows, speculative load balancing interleaves a low resolution simulation with the high resolution simulation. The low resolution simulation runs in parallel with the high resolution simulation, and ahead of the high resolution simulation by a fixed number of frames W, giving load estimates for next W frames. Thus, a low resolution simulation produces a load estimate for frame $f+W$ as the high resolution simulation advances frame f. The low resolution simulation is partitioned across the available cores on the controller node. Running the low resolution simulation in parallel has two advantages. First, it helps to mask the already small overhead of running the low resolution simulation. Second, it makes it possible to add synchronization between the low and high resolution simulation to reduce divergence. While we did not encounter divergence in the bulk fluid distribution across partitions for the simulations we ran, we expect some synchronization to be necessary for longer simulations with more turbulence; we leave a study of synchronization methods and frequency for future work.

Each partition over the high resolution simulation sends a message to the controller after every time step, indicating the simulation time to which it has advanced. If there is a new partition map for the next time step and an update to the worker assignment for that partition, the controller issues a migration command to the source worker that currently owns the partition and the destination worker which is the new owner of the partition. A partition migration is transparent to the application. After any issued migration has completed, computation over the partition proceeds to the next step.

5. Evaluation

We evaluate the benefits of distributing and load balancing sparse simulations by answering the following questions: (1) How much does distributing sparse simulations speed up simulations? (2) How well does speculative load balancing distribute work? (3) How much improvement does speculative load balancing give over static geometric partitioning and reactive load balancing?

To demonstrate the benefits of distributing and load balancing conditions, we present results for four simulations, distributed over 8 nodes using static geometric partitioning, reactive load balancing, and speculative load balancing over 8 nodes. Single-node simulations are also run as a baseline. The four simulations are:

1. A sphere drop, where a sphere of water falls into a reservoir of water, as shown in Figure 4. As the sphere gains velocity, it moves rapidly through space, testing performance with rapidly moving fluid. As the sphere drops into water and generates splashes that are under-resolved by the coarse simulation, this experiment demonstrates how a coarse simulation can generate good load estimates without synchronizing. We present results for 200 frames for this simulation.
2. A one-way dam break, where water flows out in the absence of a wall, as shown in Figure 5. This simulation exhibits a large amount of variation as the distribution of fluid over a large part of the simulation domain changes with time. We present results for 200 frames for this simulation.
3. A two-way dam break as shown in Figure 6. This simulation exhibits a large amount of variation similar to the one-way dam break. This simulation also exhibits splashes as the two bodies of water hit each other and solids, and hence demonstrates how well a coarse simulation can approximate the bulk fluid distribution in a high resolution simulation. We present results for 300 frames for this simulation.
4. A simulation with two sources of water and another water reservoir with missing walls, as shown in Figure 7. This simulation also demonstrates large variation and tests the approximation from the low resolution simulation. As the water from the two sources hit each other and the pool of water, they generate details such as thin splashes. Additionally, water flowing in from sources and the water falling out from the reservoir generate large spatial and temporal variation in the bulk distribution of fluid. We present results for 300 frames for this simulation.

5.1. Simulation and Experiment Details

All results for high resolution simulations are run over a 1024^3 grid. The experiments use micro-partitions of size $16 \times 8 \times 16$ for the simulations with two-way dam break and sources, $16 \times 16 \times 8$ for sphere drop, and $16 \times 8 \times 8$ for one-way dam break. These generate Canary tasks over $64 \times 128 \times 64$, $64 \times 64 \times 128$ and $64 \times 128 \times 128$ voxels respectively. Geometric partitioning divides the domain into $2 \times 2 \times 2$ for the sphere drop, two-way dam break and sources with falling water, and $2 \times 1 \times 4$ for the one-way dam break simulation. Speculative load balancing uses a low resolution simulation over a 128^3 grid, which is $8 \times 8 \times 8$ times smaller than the high resolution simulation. The load balancer for reactive load balancing recomputes assignments every 30 time steps of the high resolution simulation. The load balancer for speculative balancing recomputes assignments every 30 steps of the low resolution simulation.

The multi-node experiments that use geometric partitioning and reactive and speculative load balancing run the high resolution simulation over 8 worker nodes with 8 cores each, totaling to 64 cores. We use Google Cloud `n1-highmem-8` instances for the worker nodes. Each node has 8 physical cores and 52GB memory. For single-node experiments, we use a customized single 8-core instance with RAM extended to 150GB.

Both the low and high resolution simulations use 8 particles per cell and a frame rate of 30 frames per second. The high resolution simulations use a CFL number of 8. The low resolution simulations scale down the step size by using a proportionately smaller CFL number of 1. Speculative load balancing starts by first running the low resolution simulation for 30 frames and then interleaving the low and high resolution simulations from that point forward, as described in Section 4. The low resolution simulation runs on the same node as controller, over the available cores. In all our experiments, we found that the overhead for the low resolution simulation is less than 1%. This is because the computation time scales super-linearly — the time to compute a time step for the low resolution simulation is 2 orders of magnitude less than that for the high resolution simulation, the low resolution simulation requires fewer iterations for the Poisson solver, and even with the scaled down CFL, the low resolution simulation requires fewer time steps to advance the same frame time in our experiments.

The final renderings are generated from level set representations of the fluid. We generate signed distance functions from the fluid particles using OpenVDB tools to rasterize particles with a radius of 1.6 voxel units, followed by a single pass of a Gaussian filter.

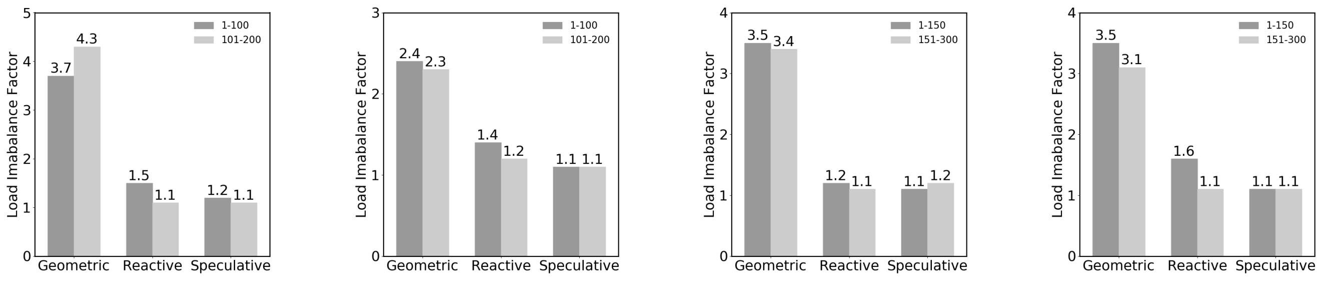
5.2. Distributed Simulations

Busy time is the total wall clock time all cores spend in compute tasks that invoke simulation functions. *Average busy time* is busy time averaged across time steps and across all cores. This is the amount of time a simulation would take if work were distributed evenly across all nodes for all time steps and there were no communication overhead. *Maximum busy time* is the maximum busy time across workers, averaged across time steps. This gives a measure of the amount of load imbalance, quantifying how well the load balancing algorithm balances load. *Total time* is the total time that a time step takes including communication overhead, and is averaged across time steps. This can vary from cluster to cluster, as the overhead is a function of the underlying network infrastructure.

Table 1 shows the average and maximum busy time and total speedup over single node for the four simulations distributed over 8 nodes, for the first half and the second half of each simulation. The total time per step on a single node for the initial 100 frames is 157s for sphere drop and 467s for the one-way dam break, and 498s and 527s for the initial 150 frames for the two-way dam break and sources. Distributing the simulations over 8 nodes with 8 cores each using static geometric partitioning speeds up the simulations by 2.5× for sphere drop, 4.1× for one-way dam break, 2.7× for two-way dam break and 2.8× for sources. The total speedup using speculative load balancing is 5.3× to 7.2×.

	Sphere Drop (1-100)			One-Way Dam (1-100)			Two-Way Dam (1-150)			Sources (1-150)		
	Ave.	Max Busy	Speedup	Ave.	Max Busy	Speedup	Ave.	Max Busy	Speedup	Ave.	Max Busy	Speedup
Single Node	153	153	**1.0**	437	437	**1.0**	455	455	**1.0**	468	468	**1.0**
Geometric	16	59	**2.5**	42	99	**4.1**	49	172	**2.7**	47	164	**2.8**
Reactive	16	24	**4.7**	41	58	**6.3**	50	62	**5.2**	50	79	**4.5**
Speculative	16	19	**5.6**	43	47	**7.2**	51	56	**6.0**	50	57	**5.3**
	Sphere Drop (101-200)			**One-Way Dam (101-200)**			**Two-Way Dam (151-300)**			**Sources (151-300)**		
	Ave.	**Max Busy**	**Speedup**	**Ave.**	**Max Busy**	**Speedup**	**Ave.**	**Max Busy**	**Speedup**	**Ave.**	**Max Busy**	**Speedup**
Single Node	145	145	**1.0**	677	677	**1.0**	545	545	**1.0**	422	422	**1.0**
Geometric	15	65	**2.2**	61	137	**4.6**	62	210	**2.7**	43	132	**3.2**
Reactive	15	16	**5.7**	57	69	**7.6**	60	64	**5.8**	48	52	**5.7**
Speculative	15	16	**5.9**	56	62	**7.9**	61	70	**5.6**	47	53	**5.5**

Table 1: *Per step average and maximum busy times in seconds and total speedup over single node for four simulations. The upper rows rows give numbers averaged over the initial frames — 1 to 100 for sphere drop and one-way dam break, and 1 to 150 for the two-way dam break and sources. The lower rows give numbers averaged over remaining 100 frames for sphere drop and one-way dam break, and the remaining 150 frames for two-way dam break and sources.*



(a) *Sphere drop* **(b)** *One-way dam break* **(c)** *Two-way dam break* **(d)** *Sources*

Figure 3: *Load imbalance factor for geometric assignment and reactive and speculative load balancing for* 1024^3 *simulations over 8 worker nodes. Load imbalance for initial frames is in dark gray and for the later frames is in light gray. Speculative load balancing performs within 20% of the ideal case for all cases.*

The total time per step on a single node is 145s and 735s for the remaining 100 frames for the sphere drop and one-way dam break, and 602s and 468s for the remaining 150 frames for the two-way dam break and sources respectively. The speedup from geometrically partitioning and distributing these simulations ranges from 2.2× to 4.6×, similar to the initial frames. Speculative load balancing gives similar speedups as before, ranging from 5.5× to 7.9×. The later frames in our examples have less temporal variation in computational load. This is because the later frames exhibit more visually interesting effects such as splashes, but have less variation in fluid distribution, and also because of the small time steps resulting from large fluid velocities and turbulence. Due to less temporal variation in computational load for later frames, reactive load balancing gives speedups similar to speculative load balancing, ranging from 5.7× to 7.6×. In cases such as these where the past is a good predictor of the future load, speculative load balancing does not provide an advantage over reactive load balancing.

Total iteration time is a function of the maximum busy time and synchronization overhead from ghost data exchanges and global reductions that are necessary for the Poisson solver. Communication overheads from global reductions are independent of how partitions are assigned to nodes, and hence, the load balancing algorithm. Synchronization overhead depends on underlying network characteristics such as latency and bandwidth, and solver efficiency, and can be reduced by using better network infrastructure and solvers. Cloud nodes have high inter-node latency; we measured a latency of 100μs on Google Cloud. Many modern clusters use high performance network interconnects such as InfiniBand [Pfi01] that provide much lower latency. Highly scalable Poisson solvers [CZY17], which is an active area of research, can further reduce the overhead.

Maximum busy time indicates possible speedups in the absence of synchronization overheads. Distributing the simulations using geometric partitioning reduces maximum busy time by 2.6× to 4.9×. Reactive load balancing reduces this further to give a net reduction of 5.9× to 9.8×. Speculative load balancing gives a total reduction of 7.8× to 10.9×. We believe that the super-linear reduction greater than 8× in busy times when distributing over 8 nodes is due to NUMA inefficiencies — the large memory on single node forces sockets to access memory on remote sockets.

5.3. Load Balancing

The *load imbalance factor* is the ratio of maximum busy time to average busy time, $\frac{\text{maximum busy time}}{\text{average busy time}}$. It indicates how well the load

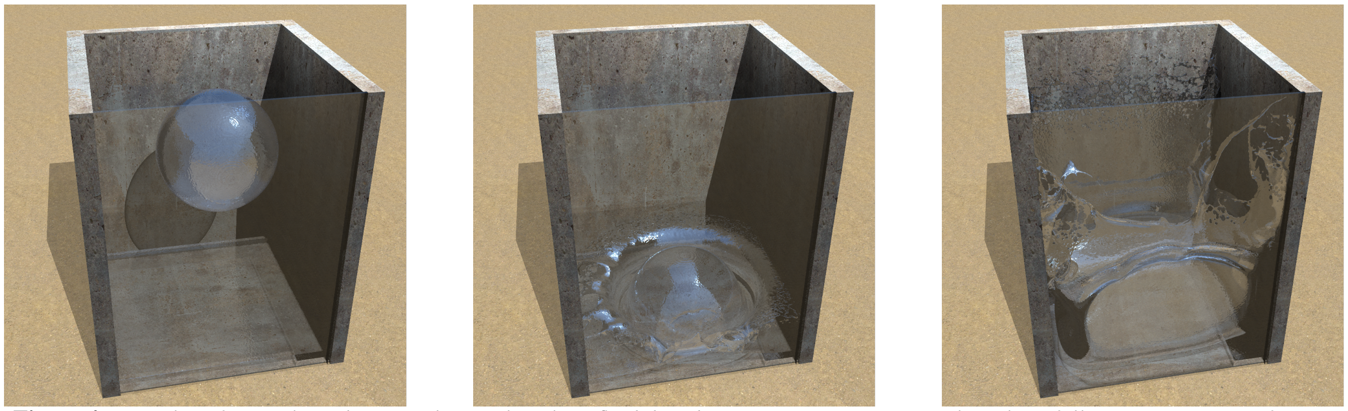


Figure 4: *Snapshots for a sphere drop simulation show how fluid distribution varies over time as the sphere falls into a reservoir of water.*

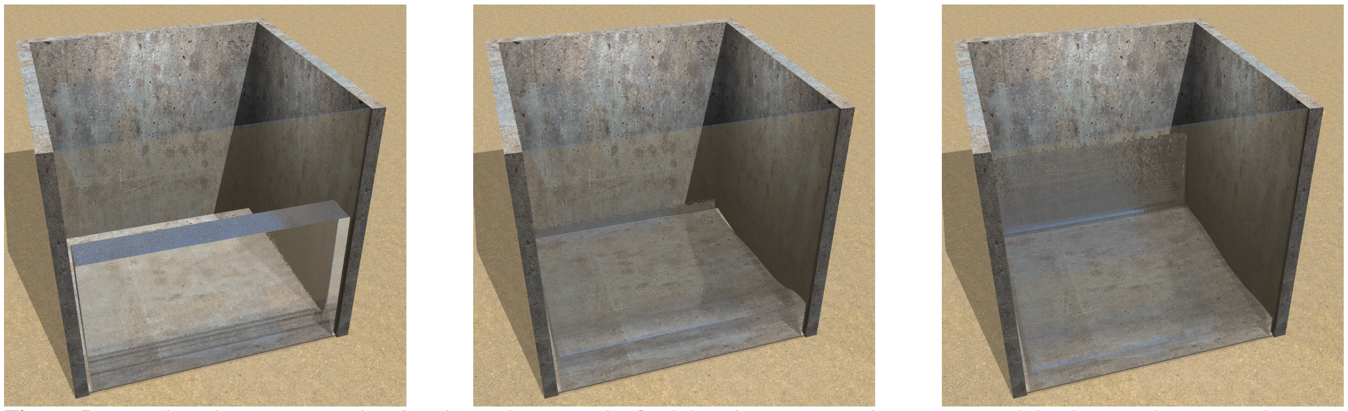


Figure 5: *Snapshots for a one-way dam break simulation — the fluid distribution over a large portion of the domain changes as the water flows out, leaving regions with water at the beginning empty and filling up empty regions.*

is balanced — a large load imbalance factor implies that there are some nodes with a lot more computational load, a load imbalance factor of one indicates a perfectly balanced simulation. Load imbalance, like maximum busy time, indicates possible speedup without synchronization overheads.

Figure 3 shows the load imbalance factor for the multi-node experiments. Static geometric partitioning performs poorly as work is unevenly distributed across nodes and this distribution changes over time. The load imbalance factor for static geometric partitioning is more than 2 for all four simulations. Load balancing reactively brings the load imbalance factor to 1.1-1.6 — within 60% the ideal case. Speculative partitioning performs with 20% of ideal for all four simulations for the entire duration of the simulation, the highest load imbalance factor being 1.2. The gains from speculative load balancing are highest during the initial frames when there is more variation in computational load across time steps.

5.4. Limitations

Speculative load balancing is useful when the low resolution simulation provides a representative estimate of load distribution across partitions over time, and when there is more temporal variation in the computational load. For the experiments we ran, speculative load balancing achieves a load imbalance within 1.2× of the ideal, without synchronization. As the included videos demonstrate, the later frames for the sphere drop, two-way dam break and sources differ in high resolution details such as thin splashes. Our current implementation does not synchronize the low resolution and high resolution simulations, which can produce slightly inaccurate estimates for the later frames. We expect synchronization to be important for more turbulent simulations and simulations run over a longer time, to provide consistent gains. Additionally, when the bulk fluid distribution over partitions varies less rapidly and the past is a good predictor of the future, speculative load balancing provides little benefit over reactive load balancing. Due to the insignificant overhead of the low resolution simulation, speculative load balancing still performs well. As Table 1 shows, reactive load balancing and speculative load balancing give similar gains over static partitioning for the later frames, when there is less variation in computational load across time steps.

6. Discussion and Future Work

Distributed fluid simulations can have greater visual detail and faster turnaround times, but automatically distributing simulations

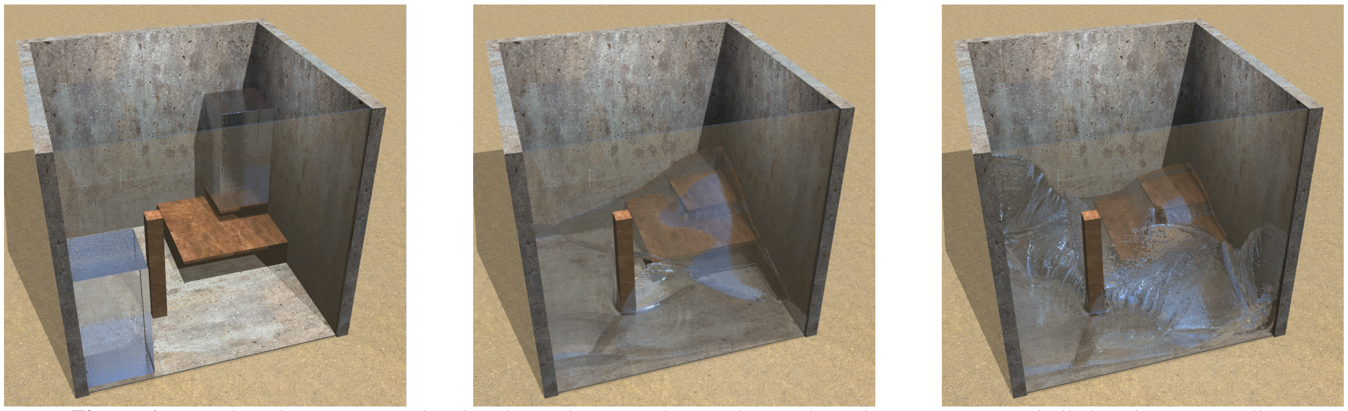


Figure 6: *Snapshots for a two-way dam break simulation — the simulation shows large variation in bulk distribution initially.*

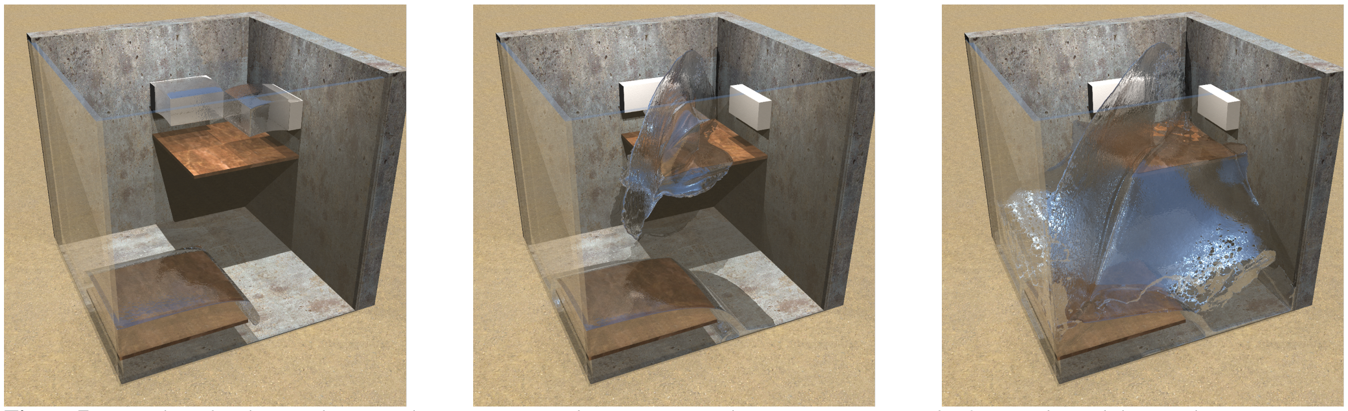


Figure 7: *Snapshots for the simulation with two sources and a reservoir — the sources are active for 2 seconds, and the simulation generates high splashes as the water from the sources hit each other and the reservoir.*

efficiently is challenging. This paper presents a system design and techniques to automatically distribute sparse fluid simulations efficiently. The key idea is speculative load balancing, which runs a low resolution simulation to predict load across partitions, and uses micro-partitions and a polynomial time algorithm to effectively distribute load. This is a general technique that can be used to load balance any simulation where a low resolution simulation provides a good estimate of future load, and is particularly useful for simulations that exhibit large temporal variation in the fluid distribution.

Experimental results for FLIP simulations show that even without synchronization, speculative load balancing performs within 20% of the ideal case when work is evenly distributed for all time steps. We believe that synchronization is important to achieve consistent gains for more complex simulations and for longer runs, but that is an area of future work. Our fluid solver implements a basic FLIP/PIC algorithm without advanced effects such as surface tension and viscosity. Exploring how speculative load balancing performs with these advanced effects and also for other simulations such as smoke and fire is another area of future work. We believe that other simulation techniques such as particle level set methods [EFFM02], narrowband FLIP [FAW*16] and mesh-based simulations [WMFB11] that selectively simulate details at a higher resolution can also benefit from speculative balancing. However, the straightforward model of using fluid cells to estimate computational load for partitions does not apply to these techniques, due to variation in the computations near and far from the surface. Developing more accurate models to estimate load for these techniques could enable them to be speculatively load balanced as well.

Acknowledgements

We would like to thank Ron Fedkiw and his research group, who have been tremendously helpful in answering questions about fluid simulations. We would also like to thank Pat Hanrahan and Intel for inviting us to be part of the Intel ISTC-VC and introducing us to the unique systems challenges that computer graphics faces. We would like to thank the Platform Lab at Stanford for their support and feedback. We are thankful to the reviewers for their comments and suggestions that were very helpful in improving this paper.

This work was funded by the National Science Foundation (CSR grant #1409847) and conducted in conjunction with the Intel Science and Technology Center - Visual Computing. The experiments were made possible by a generous grant from the Google Cloud Platform Education Grants program.

References

[AGL*17] Aanjaneya M., Gao M., Liu H., Batty C., Sifakis E.: Power diagrams and sparse paged grids for high resolution adaptive liquids. *ACM Transactions on Graphics (TOG) 36*, 4 (2017), 140. 2

[BB87] Berger M. J., Bokhari S. H.: A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 5 (1987), 570–580. 2

[BDF*07] Boman E., Devine K., Fisk L. A., Heaphy R., Hendrickson B., Vaughan C., Catalyurek U., Bozdag D., Mitchell W., Teresco J.: Zoltan 3.0: parallel partitioning, load-balancing, and data management services; userâĂŹs guide. *Sandia National Laboratories, Albuquerque, NM* (2007). 2, 3

[Ble18] Blender. `https://www.blender.org/`, 2018. 2

[BOVVDZ16] Bansal N., Oosterwijk T., Vredeveld T., Van Der Zwaan R.: Approximating vector scheduling: almost matching upper and lower bounds. *Algorithmica 76*, 4 (2016), 1077–1096. 3

[BSL*16] Bernstein G. L., Shah C., Lemire C., Devito Z., Fisher M., Levis P., Hanrahan P.: Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics (TOG) 35*, 2 (May 2016), 21:1–21:12. URL: `http://doi.acm.org/10.1145/2892632`, `doi:10.1145/2892632`. 3

[BTSA12] Bauer M., Treichler S., Slaughter E., Aiken A.: Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, IEEE Computer Society Press, pp. 66:1–66:11. URL: `http://dl.acm.org/citation.cfm?id=2388996.2389086`. 3

[BVK09] Becker A., Venkataraman R., Kale L.: Patterns for overlapping communication and computation. *Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL* (2009). 3, 4

[CBD*07] Catalyurek U. V., Boman E. G., Devine K. D., Bozdag D., Heaphy R., Riesen L. A.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (March 2007), pp. 1–11. `doi:10.1109/IPDPS.2007.370258`. 2

[Chr10] Christensen B. B.: Efficient Algorithms for Controllable Fluid Simulations and High-Resolution Level Set Deformations. 2

[CK04] Chekuri C., Khanna S.: On multidimensional packing problems. *SIAM journal on computing 33*, 4 (2004), 837–851. 3

[CZY17] Chu J., Zafar N. B., Yang X.: A schur complement preconditioner for scalable parallel fluid simulation. *ACM Transactions on Graphics (TOG) 36*, 5 (2017), 163. 8

[DC*96] Desbrun M., Cani M.-P., et al.: Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics workshop on Computer animation and simulation* (1996), vol. 96, Springer, pp. 61–76. 2

[DHF*11] Dubey P., Hanrahan P., Fedkiw R., Lentine M., Schroeder C.: Physbam: Physically based simulation. In *ACM SIGGRAPH 2011 Courses* (2011), ACM, p. 10. 2

[DJP*11] DeVito Z., Joubert N., Palacios F., Oakley S., Medina M., Barrientos M., Elsen E., Ham F., Aiken A., Duraisamy K., et al.: Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), SC'11, ACM, p. 9. 3

[DM98] Dagum L., Menon R.: OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering 5*, 1 (1998), 46–55. 2

[EFFM02] Enright D., Fedkiw R., Ferziger J., Mitchell I.: A hybrid particle level set method for improved interface capturing. *Journal of Computational physics 183*, 1 (2002), 83–116. 2, 10

[EQYF13] English R. E., Qiu L., Yu Y., Fedkiw R.: Chimera grids for water simulation. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2013), ACM, pp. 85–94. 2

[FAW*16] Ferstl F., Ando R., Wojtan C., Westermann R., Thuerey N.: Narrow band FLIP for liquid simulations. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 225–232. 2, 10

[FLR98] Frigo M., Leiserson C. E., Randall K. H.: The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (1998), PLDI '98, ACM, pp. 212–223. URL: `http://doi.acm.org/10.1145/277650.277725`, `doi:10.1145/277650.277725`. 2

[FSJ01] Fedkiw R., Stam J., Jensen H. W.: Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 15–22. 2

[GBW16] Goldade R., Batty C., Wojtan C.: A practical method for high-resolution embedded liquid surfaces. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 233–242. 2

[GM77] Gingold R. A., Monaghan J. J.: Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society 181*, 3 (1977), 375–389. 2

[Gra69] Graham R. L.: Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics 17*, 2 (1969), 416–429. 4

[Har62] Harlow F. H.: *The particle-in-cell method for numerical solution of problems in fluid dynamics*. Tech. rep., Los Alamos Scientific Lab., N. Mex., 1962. 2

[HBE98] Hu Y., Blake R. J., Emerson D. R.: An optimal migration algorithm for dynamic load balancing. *Concurrency and Computation: Practice and Experience 10*, 6 (1998), 467–483. 3

[HMB12] Humphrey A., Meng Q., Berzins M.: The Uintah framework: A unified heterogeneous task scheduling and runtime system. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (2012), IEEE, pp. 2441–2448. 3

[Hou18] Houdini. `http://www.sidefx.com/docs/houdini/index.html`, 2018. 2

[HW65] Harlow F. H., Welch J. E.: Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The physics of fluids 8*, 12 (1965), 2182–2189. 2

[JSS*15] Jiang C., Schroeder C., Selle A., Teran J., Stomakhin A.: The affine particle-in-cell method. *ACM Transactions on Graphics (TOG) 34*, 4 (2015), 51. 2

[Kar03] Karypis G.: Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (2003), SC '03, ACM, pp. 56–. URL: `http://doi.acm.org/10.1145/1048935.1050206`, `doi:10.1145/1048935.1050206`. 2

[KHAL*14] Kaiser H., Heller T., Adelstein-Lelbach B., Serio A., Fey D.: HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (2014), PGAS '14, ACM, pp. 6:1–6:11. URL: `http://doi.acm.org/10.1145/2676870.2676883`, `doi:10.1145/2676870.2676883`. 3

[KK93] Kale L. V., Krishnan S.: CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices* (1993), vol. 28, ACM, pp. 91–108. 1, 2, 3

[KK96] Karypis G., Kumar V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (1996), Supercomputing '96, IEEE Computer Society. URL: `http://dx.doi.org/10.1145/369028.369103`, `doi:10.1145/369028.369103`. 2

[KKRK*16] Kjolstad F., Kamil S., Ragan-Kelley J., Levin D. I., Sueda S., Chen D., Vouga E., Kaufman D. M., Kanwar

G., Matusik W., et al.: Simit: A language for physical simulation. *ACM Transactions on Graphics (TOG) 35*, 2 (2016), 20. 3

[KT06] Kleinberg J., Tardos E.: *Algorithm design*. Pearson Education India, 2006. 2, 4

[LCPF12] Lentine M., Cong M., Patkar S., Fedkiw R.: Simulating free surface flow with very large time steps. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2012), Eurographics Association, pp. 107–116. 1, 2

[LGF04] Losasso F., Gibou F., Fedkiw R.: Simulating water and smoke with an octree data structure. In *ACM Transactions on Graphics (TOG)* (2004), vol. 23, ACM, pp. 457–462. 2

[LKK14] Lifflander J., Krishnamoorthy S., Kale L. V.: Optimizing data locality for fork/join programs using constrained work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), SC '14, IEEE Press, pp. 857–868. URL: http://dx.doi.org/10.1109/SC.2014.75, doi:10.1109/SC.2014.75. 2

[May18] Maya. https://www.autodesk.com/products/maya/overview, 2018. 2

[MLJ*13] Museth K., Lait J., Johanson J., Budsberg J., Henderson R., Alden M., Cucka P., Hill D., Pearce A.: OpenVDB: an open-source data structure and toolkit for high-resolution volumes. In *Acm siggraph 2013 courses* (2013), ACM, p. 19. 1, 2, 3

[MQSL17] Mashayekhi O., Qu H., Shah C., Levis P.: Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 513–526. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/mashayekhi. 3

[MSQ*18] Mashayekhi O., Shah C., Qu H., Lim A., Levis P.: Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud. *To appear in ACM Transactions on Graphics (TOG)* (2018). 1, 3

[NB11] Nielsen M. B., Bridson R.: Guide shapes for high resolution naturalistic liquid simulation. *ACM Transactions on Graphics (TOG) 30*, 4 (2011), 83. 2

[NCZ*09] Nielsen M. B., Christensen B. B., Zafar N. B., Roble D., Museth K.: Guiding of smoke animations through variational coupling of simulations at different resolutions. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2009), ACM, pp. 217–226. 2

[OB98] Oliker L., Biswas R.: PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing 52*, 2 (1998), 150–177. 3

[OR94] Ou C.-W., Ranka S.: Parallel incremental graph partitioning using linear programming. In *Supercomputing'94., Proceedings* (1994), IEEE, pp. 458–467. 3

[PB94] Pilkington J., Baden S.: Partitioning with spacefilling curves. *CSE Technical Report CS94-349* (1994). 2

[Pfi01] Pfister G. F.: An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O 42* (2001), 617–632. 8

[PGDS*14] Pearce O., Gamblin T., De Supinski B. R., Arsenlis T., Amato N. M.: Load balancing n-body simulations with highly non-uniform density. In *Proceedings of the 28th ACM international conference on Supercomputing* (2014), ACM, pp. 113–122. 1, 2, 3

[Phe08] Pheatt C.: Intel® threading building blocks. *Journal of Computing Sciences in Colleges 23*, 4 (2008), 298–298. 2

[QMSL18] Qu H., Mashayekhi O., Shah C., Levis P.: Decoupling the Control Plane from Program Control Flow for Flexibility and Performance in Cloud Computing. In *To appear in European Conference on Computer Systems (Eurosys)* (2018). 3, 5, 6

[RTWT12] Raveendran K., Thuerey N., Wojtan C., Turk G.: Controlling liquids using meshes. In *Proceedings of the 11th ACM SIGGRAPH/Eurographics conference on Computer Animation* (2012), Eurographics Association, pp. 255–264. 2

[SABS14] Setaluri R., Aanjaneya M., Bauer S., Sifakis E.: SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG) 33*, 6 (2014), 205. 1, 2

[SKK00] Schloegel K., Karypis G., Kumar V.: *Graph partitioning for high performance scientific simulations*. Army High Performance Computing Research Center, 2000. 2, 3

[SLT*15] Slaughter E., Lee W., Treichler S., Bauer M., Aiken A.: Regent: A high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2015), SC'15, ACM, p. 81. 3

[Sni98] Snir M.: *MPI–The Complete Reference: The MPI Core*, vol. 1. MIT press, 1998. 1, 2, 3

[Sta99] Stam J.: Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 121–128. 2

[TP16] Thuerey N., Pfaff T.: Mantaflow.(2016), 2016. 2

[WMFB11] Wojtan C., Müller-Fischer M., Brochu T.: Liquid simulation with mesh-based surface tracking. In *ACM SIGGRAPH 2011 Courses* (2011), ACM, p. 8. 2, 10

[ZB05] Zhu Y., Bridson R.: Animating sand as a fluid. *ACM Transactions on Graphics (TOG) 24*, 3 (2005), 965–972. 2, 5