DOI: 10.1111/cgf.13809

COMPUTER GRAPHICS *forum*
Volume 39 (2020), number 1 pp. 375–388

Accelerating Distributed Graphical Fluid Simulations with Micro-partitioning

Hang Qu, Omid Mashayekhi, Chinmayee Shah and Philip Levis

Department of Electrical Engineering and Department of Computer Science, Stanford University, Stanford, CA, USA
quhang@stanford.edu, omidmsu@gmail.com, chshah@stanford.edu, pal@cs.stanford.edu

Abstract

Graphical fluid simulations are CPU-bound. Parallelizing simulations on hundreds of cores in the computing cloud would make them faster, but requires evenly balancing load across nodes. Good load balancing depends on manual decisions from experts, which are time-consuming and error prone, or dynamic approaches that estimate and react to future load, which are non-deterministic and hard to debug.

This paper proposes Birdshot scheduling, an automatic and purely static load balancing algorithm whose performance is close to expert decisions and reactive algorithms without their difficulty or complexity. Birdshot scheduling's key insight is to leverage the high-latency, high-throughput, full bisection bandwidth of cloud computing nodes. Birdshot scheduling splits the simulation domain into many micro-partitions and statically assigns them to nodes randomly. Analytical results show that randomly assigned micro-partitions balance load with high probability. The high-throughput network easily handles the increased data transfers from micro-partitions, and full bisection bandwidth allows random placement with no performance penalty. Overlapping the communications and computations of different micro-partitions masks latency.

Experiments with particle-level set, SPH, FLIP and explicit Eulerian methods show that Birdshot scheduling speeds up simulations by a factor of 2-3, and can out-perform reactive scheduling algorithms. Birdshot scheduling performs within 21% of state-of-the-art dynamic methods that require running a second, parallel simulation. Unlike speculative algorithms, Birdshot scheduling is purely static: it requires no controller, runtime data collection, partition migration or support for these operations from the programmer.

Keywords: distributed graphics, hardware, fluid modelling, animation

ACM CCS: • Computing methodologies → Distributed computing methodologies; Distributed simulation; Computer graphics

1. Introduction

Fluid simulations are a cornerstone in making cinematic special effects. Simulating those spectacular scenes, such as stormy seas, sudden floods or water falls, requires intensive computations. A few seconds of screen time can take hours or days to simulate on a single node.

Graphical fluid simulations are CPU-bound: parallelizing them across more cores runs them faster. Owning and maintaining a 1000-core cluster, however, is expensive, and the cluster may not keep high utilization due to the long delays of having artists in the loop. On-demand cloud platforms, in contrast, charge on a node per hour basis, such that users only pay for the resources they use. Furthermore, the fungibility of time and parallelism means that parallelizable applications can run faster at the same price. Running a simulation 10 times faster on 10 times more nodes costs the same but completes an order of magnitude faster. Recent work has shown that single-threaded complex simulations can be automatically distributed to run on over a thousand cores in the cloud, drastically speeding up simulations and increasing their details [MSQ*18].

The decision of how to partition a simulation domain across nodes and cores (also called domain decomposition) has a tremendous effect on performance. Each simulation step is limited by the speed of the slowest node. A poor partitioning can place all of the required computation on a few nodes while the majority of the nodes sit idle. In contrast, if the work were evenly partitioned, the slowest node has only a small fraction of the work and the simulation runs faster.

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

Partitioning is especially difficult in graphical fluid simulations because the computation needed varies over both spatial and temporal dimensions. In particle-level set simulations, for example, the narrow band of cells containing the interface between fluid and air requires the greatest computation due to a high density of particles. Fluid cells require significant but lesser computation because the computation is purely Eulerian, while air cells require no computation at all. Furthermore, because the fluid and interface move, exactly which cells are interface, fluid or air changes over time.

One partitioning approach to evenly distribute work across nodes is to dynamically change how the simulation domain is partitioned and assigned to nodes, i.e. dynamic load balancing. When most computation happens on a tiny portion (less than 10%) of the entire domain, this approach gives an order of magnitude speedup over static and uniform partitioning. For example, a speculative load balancing algorithm [SHQL18] runs a low-resolution fluid simulation alongside the actual one in order to estimate load distribution, uses the estimate to decide how to assign partitions and achieves 5–8 times speedup over static and uniform partitioning.

Two drawbacks of dynamic load balancing algorithms are increased system complexity and non-deterministic runtime behaviour. Dynamic load balancing algorithms require code to migrate partitions, maintain an index of current partition locations and dynamically gather load data with which to make load balancing decisions. As these decisions are based on runtime performance and timing, two runs of the same simulation may distribute and execute differently: tracing execution to find bugs across the distributed system is time-consuming.

This paper proposes a new scheduling approach for distributed graphical fluid simulations, called Birdshot scheduling. Birdshot scheduling performs almost as well as state-of-the-art dynamic scheduling algorithms based on speculative execution [SHQL18], but has no control overhead, does not need to dynamically migrate data and generates a deterministic, repeatable schedule. Birdshot scheduling works on simulations with an underlying Eulerian geometry, such as uniform grids and sparse data structures, like OpenVDB [Mus13] or sparse paged grids [SABS14]. The key technique in Birdshot scheduling is to *micro-partition* the simulation, i.e. split the domain into many fine-grained *partitions* (usually 4–64 times the total number of cores), and randomly assign micro-partitions to cores. This has two principal benefits. First, random assignment balances load with high probability as long as enough partitions are used. The exact partition-to-core ratio depends on the workload distribution and we derive analytic solutions for the recommended ratio. Second, micro-partitioning helps each core to efficiently handle the computation and communication of multiple partitions, allowing them to overlap the computation of one partition with the communication time of another. Barrier operations cannot perform such overlapping and require different optimizations. The end result is a set of scheduling policies, called Birdshot scheduling, that automatically balances load and masks communication for micro-partitioned simulations.

Experiments show that Birdshot scheduling runs simulations 2–3× faster than static and uniform partition assignments and scales to run on over 1000 cores. In addition, Birdshot scheduling can in some cases outperform dynamic load balancing algorithms. In a FLIP simulation, Birdshot scheduling is only 21% slower than a state-of-art dynamic load balancing algorithm [SHQL18], but it is much simpler.

Birdshot scheduling is general enough to work on a wide range of fluid simulation methods, including Eulerian, Lagrangian and hybrid methods. Experimental results show that simulations using Birdshot scheduling scale well to hundreds of cores. We find that Birdshot scheduling's scalability can be limited by barriers: they do not allow overlapping computation with communication as every micropartition must complete before any one can move forward. Furthermore, certain simulation methods scale poorly to large numbers of partitions. For example, the performance of most Poisson solver preconditioners can degrade significantly as the number of partitions increases.

The key benefit of Birdshot scheduling is its simplicity. Although its schedule is static and repeatable, it performs almost as fast as dynamic load balancing algorithms. It can be used in any simulation implementation that supports static distributed execution: it requires neither migration nor load balancing logic, simplifying debugging and development.

This paper makes four research contributions:

1. Introducing micro-partitioning as a technique to balance load and mask communication time in distributed fluid simulations running on the computing cloud.
2. Analysing how well randomized assignment of micro-partitions balances load depending on the number of nodes, the number of partitions and the work skew between partitions.
3. Presenting how to overlap communication and computation to reduce execution time and proposes a novel user-level TCP communication library designed to address the barrier operation performance bottlenecks.
4. Evaluating the proposed techniques on particle-level set, smoothed-particle hydrodynamics (SPH), pure Eulerian and FLIP simulations, finding that using Birdshot runs 2.0–3.3× faster than static geometric partitioning, can out-perform reactive scheduling and performs within 21% of state-of-the-art speculative execution methods.

The source code of Birdshot scheduling algorithm and the system to run the applications is open source and freely available for use at https://sing.stanford.edu/nimbus/birdshot.zip.

2. Related Work

This section reviews related work in graphical fluid simulations and discusses relevant system techniques.

2.1. Fluid simulations

Graphical fluid simulations use varied data structures and numerical methods. SPH [GM77, DC*96] models fluid as particles. Grid-based methods typically use a MAC grid discretization [HW65], the semi-Lagrangian advection scheme [Sta99] and a pressure Poisson solver for incompressibility [FSJ01]. More recent approaches use a hybrid of particles and grids. For example, the particle-in-cell method [Har62, JSS*15] uses particles for advection and grids

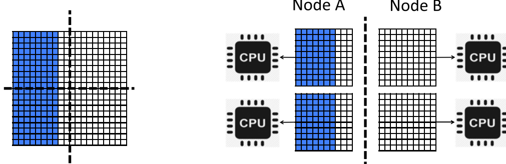
© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

for other physical values. The particle-level set method [EFFM02] captures finer details near the fluid–air interface by adding marker particles to the grid.

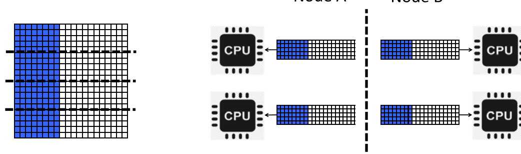
Capturing more visual details in fluid simulations requires more particles or a finer grid resolution, causing more computations per simulation frame. Adaptive data structures [Mus13, SABS14] and chimera grids [EQYF13] mitigate the performance problem by only refining the grid where visual details are important. This paper explores an orthogonal approach that enables a fluid simulation to compute faster.

In production, a large fluid simulation is often split into smaller independent simulations for parallel execution. For example, [Whi12, RMW*16] independently run several coarse and finer fluid simulations on multiple nodes, stitch them together and manually fix the mismatches at simulation boundaries.

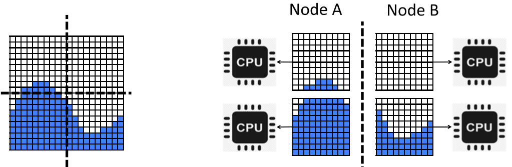
Fluid simulations can leverage distributed and heterogeneous platforms, by distributing sparse data structures on clusters [BBAW15] and GPUs [WTYH18], distributing solvers [LMAS16] on GPUs and distributing material point methods on GPUs [GWW*18].



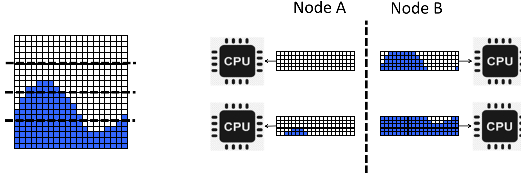
(a) The starting frame (square partitioning).



(b) The starting frame (horizontal partitioning).



(c) A frame during the simulation (square partitioning).



(d) A frame during the simulation (horizontal partitioning).

Figure 1: *The figures illustrate two frames in a 2D fluid simulation, where the blue cells represent fluid and the white cells represent air. Balancing the computation work on cores for faster execution is hard due to spatial and temporal load variance.*

2.2. System techniques

Load balancing algorithms. There is a rich literature of load balancing algorithms in scientific computing, which can be categorized as static or dynamic. Static load balancing assigns application data to nodes before an application executes and does not change the assignment during runtime. It is commonly used in graph processing, where the load distribution is static and predictable [KK96, CBD*07, Kar03].

Dynamic load balancing migrates application data between nodes to adjust the load distribution during runtime. Work stealing immediately reacts to an idle core by fetching work from cores on the same node [FLR98] or remote nodes [LKK14], but is prone to repeatedly rebalancing load between nodes [LKK14]. Centralized dynamic load balancing algorithms [NH85] can bottleneck at the central node [MK13]. Fully distributed load balancing algorithms [XLD97, MK13] solve the scalability problem but converge slowly. Hierarchical load balancing algorithms [ZBMK11, JMM*13] adopt a multi-level scheduling architecture, where lowest level schedulers balance loads within a group of nodes, and higher level schedulers balance loads between lower level schedulers.

Charm++ [AGJ*14] is a flexible simulation framework that provides mechanisms to implement load balancing algorithms. Charm++ software distribution includes many algorithm implementations, including the classic reactive load balancing (called GreedyLB in Charm++) we used in the evaluation. Birdshot scheduling is orthogonal to Charm++; it is a new algorithm that could be implemented in that framework.

Fine-grained computing. The idea of splitting an application into fine-grained computation units is used in both cloud computing and scientific computing. For example, [OPR*13, OWZS13] propose using finer computation units for faster failure recovery and resource preemption in the computing cloud. [AGJ*14] proposes running more processes than the number of cores to mask communication time in scientific computing applications. This paper introduces the similar idea to distributed graphical fluid simulations and analyses its relation with load balancing and communication performance.

Runtime support for overlapping computation and communication. Most existing distributed computing frameworks provide the mechanism to overlap computation and communication. MPI [url17] provides asynchronous communication primitives (e.g. `MPI_Isend` and `MPI_Irecv`) for a user to control what computations to run during an ongoing communication. In Charm++ [AGJ*14], computations are triggered by messages, and a computation blocked by messages is not scheduled. Task-based frameworks, such as Legion [BTSA12], HPX [KHAL*14] and Uintah [HMB12], model computations as tasks that read or modify data objects. The runtime figures out runnable tasks to overlap with communication time. This paper proposes that micro-partitioning

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

helps make the overlapping mechanism effective in distributed fluid simulations.

3. Problem Statement

Distributing a simulation improves performance by using more cores. An implicit assumption in this improvement is that the computational load is balanced across those cores. Using twice as many cores can double simulation speed, but if half of the cores are idle, then those performance gains will not be realized. For this reason, *partitioning*, the decision of how to break a simulation into smaller parts and assign them to cores, is critical to achieving speedups from distribution.

Consider the example in Figure 1, which shows a simple depiction of a dam break fluid simulation. The partitions on the left contain more fluid initially. As the simulation evolves, the distribution of fluid across the partitions changes and the fluid moves to the lower partitions. If the simulation domain is split into four square partitions (Figure 1a), the two right partitions contain no fluid and have no computation work: the simulation is run on 4 cores, but only 2 of the cores are used. In contrast, splitting the domain into four horizontal bands evenly balances the load across all 4 cores (Figure 1b): a horizontally partitioned simulation will run twice as fast as a square partitioned one.

3.1. Spatial and temporal load variance

Fluid simulations are difficult to partition because their computational load varies over both space and time. Simulated fluids typically do not occupy the entire domain: waves, smoke and fire are all visually interesting because of the surface where they meet the air, called the *interface*. Furthermore, different regions of the simulation have different computational requirements. In a standard particle-level set simulation, for example, air cells require no computation, fluid cells require solving Navier–Stokes and cells close to the interface require additional computation to manage particles. As a result, fluid cells near the boundary require the most computation, followed by fluid cells far from the boundary, air cells close to the boundary and finally air cells far from the boundary require no computation at all.

If a simulation is spatially partitioned across cores, some cores will have much more work than others, and the simulation will only run as fast as the slowest core. For example, in Figure 1(c), the core simulating the bottom left partition bottlenecks the simulation. It is responsible for simulating 95 fluid cells, while the average number of fluid cells per core is 37. If the computation work were evenly distributed to cores, the simulation would run approximately 95/37=2.6× faster. The computational load of the simulation varies over space, such that most simple partitioning strategies lead to poor load balancing across cores.

Furthermore, a simple partitioning that works well for a particular timestamp may not work well for other times. As the simulation evolves over time, the fluid shape changes, and so does the amount of computation in each partition. For example, horizontal partitioning evenly distributes computation work to cores for the starting frame in Figure 1(b). As the simulation evolves to Figure 1(d), the same partitioning has poor load balancing, so runs much slower than other partitionings (e.g. vertical bands).

One approach to balance load is to dynamically change how partitions are assigned to nodes. But one major obstacle is the complexity of debugging. Debugging distributed applications is hard because application states are distributed across multiple nodes [GMGK84, CBM90]. Dynamic load balancing makes debugging even harder because the user does not control which node stores which state. For example, debugging the computation on a cell requires looking at its neighbouring cells, but those neighbouring cells can be on a remote node. The remote node is hard to identify because it changes as partitions are migrated.

3.2. Inter-partition communication

Most computation on a cell requires the values from its neighbouring cells. Distributed simulations involve data transfers between neighbouring partitions. For example, a simulation performs such data transfer before advecting velocity to ensure the advection computation on every cell reads the most up-to-date velocity data.

Data transfers between partitions can greatly slow down execution. If a computation over a partition needs data from a previous computation over a neighbouring partition, it cannot start till the transfer completes. Parallelizing a simulation across many cores can exacerbate this bottleneck, as the ratio of communication to computational time increases (parallelization across more cores means more partition surface and more data transferred, but the total amount of computation remains the same).

A careful design around cloud networking performance helps mitigate this data transfer problem. Section 5.3 gives more detailed analysis. In short, fluid simulations will benefit from the high, full-bisection bandwidth of the cloud if they break geometric locality and overlap data transfer with computation. First, geometric locality (i.e. assigning communicating partitions to neighbouring nodes) matters in supercomputers, where certain nodes communicate faster than others [JMM*13, ARK10]. In contrast, the cloud network has uniform performance between different nodes. As long as two partitions are not assigned to the same node, the communication performance is similar no matter which nodes are used. Consequently partition assignment is no more restricted by geometric locality. Second, in order to mask communication time, a simulation needs to break communication into small units to overlap with computation, causing slightly more data transfer. The cloud network handles this well due to sufficient bandwidth.

4. Partitioning in Birdshot Scheduling

Birdshot's partitioning approach has two key points. First, it micropartitions the domain into many (e.g. in the workloads we examine 4–32) partitions per core. Having enough partitions is essential for Birdshot to balance load as explained in Section 5. Second, the number of partitions assigned to a node is proportional to its number of cores, but which partition is assigned to which node is generated *randomly*. This naturally spreads computationally intensive partitions across nodes.

© 2019 The Authors. Computer Graphics Forum published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

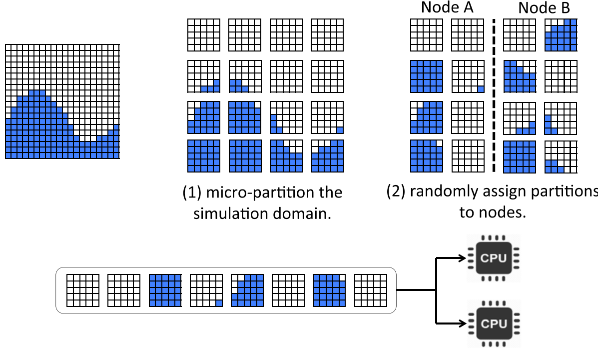


Figure 2: *How Birdshot scheduling partitions the simulation domain across nodes and cores. The example illustrates a 2D fluid simulation, where blue cells are fluid and white cells are air.*

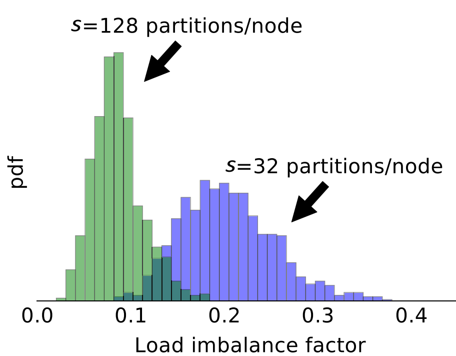


Figure 3: *The distribution of the load imbalance factor when using 32 or 128 partitions per node ($n = 32$, $p = 0.6$). Having more partitions makes the load imbalance factor more close to zero.*

Figure 2 shows an example of how Birdshot partitions a simulation across 2 nodes and 4 cores. The simulation domain is split into 16 partitions (4× the total number of cores) with 25 cells each. The nodes have the same number of cores, so they are assigned the same number of partitions. The mapping between partitions and nodes is generated randomly and does not change during execution. This random assignment means the two nodes are expected to have a similar amount of work. In the shown frame, the first node has 71 blue cells, while the second node has 79.

Randomized assignment of micro-partitions balances load across nodes, but the runtime balances load across cores of the same node. A static mapping between partitions and cores may overload a core, because the core cannot offload its work to the other cores on the same node even if they are idle. So, Birdshot's runtime dynamically maps partitions to cores: the computation work of partitions is dispatched to the next available core on the node. The dynamic mapping does not cause substantial overhead because those cores share the same memory. Figure 2 illustrates this execution model. The computation of a partition depends on data transferred from other partitions, so a node tracks the data dependency of each assigned partition (i.e. what simulation steps on the partition have received all data and thus ready to run), queues the ready computations and the next idle core fetches computations from the queue to execute them in a first-in-first-out manner.

5. Balancing Load with Micro-partitioning

This section mathematically analyses why having more randomly assigned partitions balances load better across nodes and how many partitions are needed, and then discusses corner cases and how to choose the number of partitions.

5.1. Model specification

This subsection specifies the mathematical model used to analyse how well Birdshot scheduling balances the computational load between nodes. The model assumes the amount of computation work in each partition follows a binary distribution. Approximating all partitions as one of the two extreme points makes the modelled load distribution more skew than the real distribution, so the model gives a worst-case bound of how well Birdshot scheduling balances load.

Consider a simulation that runs on n nodes and is split into $s \cdot n$ partitions, where s is the number of partitions per node. The model analyses a short time period of the execution, e.g. one iteration of the simulation, and assumes the load distribution is static in the time period. To reflect the binary load distribution assumption, the model assumes p percentage of partitions have the same amount of computation work in the time period, while the other $1-p$ percentage have no computation work. Note that Birdshot scheduling itself does not know *a priori* whether partitions have computation work or not and works under dynamic load distribution.

Ideally, load would be perfectly balanced if every node got $p \cdot s$ busy partitions that have computation work and $(1-p)\cdot s$ idle partitions with no computation work. However, due to the randomness, a particular node may receive more than $p \cdot s$ busy partitions: the more busy partitions the most overloaded node receives, the worse the load imbalance. The load imbalance factor (LIF) is defined as the ratio of how much more computation work is assigned to the most overloaded node than the average. In this model, the load imbalance factor can be written as:

$$LIF = \frac{\textit{\# of busy partitions on most overloaded node}}{\textit{Average \# of busy partitions per node}} - 1. \quad (1)$$

For example, if there are 10 busy partitions per node in average and the most overloaded node is assigned 15 busy partitions, LIF equals to $15/10 - 1 = 0.5$. If every node gets the same amount of computation work, LIF becomes zero.

In this model, the expectation of LIF can be approximated by a function depending on the percentage of busy partitions (p), the number of nodes (n) and the number of partitions per node (s):

$$E(LIF)|_{p,s,n} \approx \sqrt{2\left(\frac{1}{p}-1\right)}\sqrt{\frac{\log n}{s}}. \quad (2)$$

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

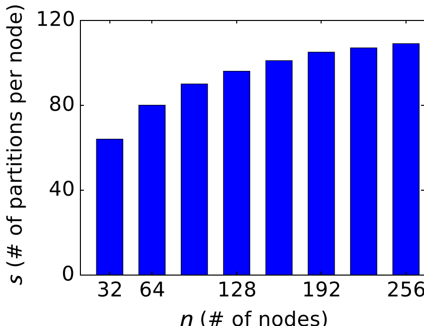


Figure 4: *To keep the expectation of the load imbalance factor fixed, the number of partitions per node depends on the number of nodes (n) ($p = 0.6$, $E(LIF) = 0.21$). The curve is logarithmic.*

We refer those who want to know how Equation (2) is derived to the Appendix.

5.2. Implications

This subsection describes two implications from Equation (2).

First, a sufficient number of partitions balances the computation work between nodes well. Imagine a simulation running on 4 nodes with half busy partitions and half idle partitions. If 4 partitions are created, two nodes will have no work to do. If 8 partitions are created, the probability that both partitions on a node are idle will be less than 25%. If 64 partitions are created, the computation work of a node will be the average of 16 randomly chosen partitions and more likely to be balanced.

Equation (2) indicates that keeping the number of nodes (n) and the percentage of busy partitions (p) fixed, increasing the number of partitions per node (s) can make the expectation of the load imbalance factor arbitrarily small. Figure 3 shows that the load imbalance factor moves towards zero with more partitions per node.

Second, balancing load on more nodes needs more partitions, and quantitatively, the number of partitions per node should grow logarithmically with the number of nodes. This indicates the scalability of Birdshot scheduling is limited by the number of partitions a node can hold.

The result is derived from Equation (2): to keep the same load imbalance factor, the number of partitions per node (s) should increase logarithmly as the number of nodes (n) increases, assuming the percentage of busy partitions (p) is fixed. When using more nodes, it is more likely that one of the nodes gets too many busy partitions, so the expectation of the load imbalance factor will increase. Adding slightly more partitions per node offsets the increased imbalance as shown in Figure 4.

5.3. Discussion

Why not migrate partitions between nodes? Birdshot scheduling never migrates a partition between nodes during execution, because it balances load well enough such that migrating partitions cannot substantially improve the balance, only adding communication overhead. First, Birdshot may balance load poorly due to randomness, but the probability is low. As the load distribution changes over time, the low probability case does not last long and its performance impact is averaged out across the entire execution.

Second, we experimentally verify that migrating partitions provides only a small performance improvement by evaluating a dynamic variation of Birdshot. The algorithm builds on top of Birdshot's random partition assignment, and decides when and what partitions to migrate by monitoring the CPU cycles used by each partition and the CPU utilization of each node. Both statistics are smoothened by averaging with an exponential window function. If the CPU utilization of a node is higher than a threshold (e.g. 10%) over the average, Birdshot will swap the busiest partition of the most overloaded node with the least busy partition on the most underloaded node. Every two swaps are separated by a time interval (e.g. 60 s) to avoid oscillations.

The evaluation runs on two simulations, PARSEC and Lassen (see Section 7.1). The new algorithm speeds up PARSEC only by 9% but cannot speed up Lassen even if the algorithm parameters (e.g. the CPU utilization threshold and the swapping interval) are well tuned. This is because Lassen's load distribution changes fast, and partition migration must happen more often to balance load well. But more migrations cause more data migration overhead that offsets the performance benefit.

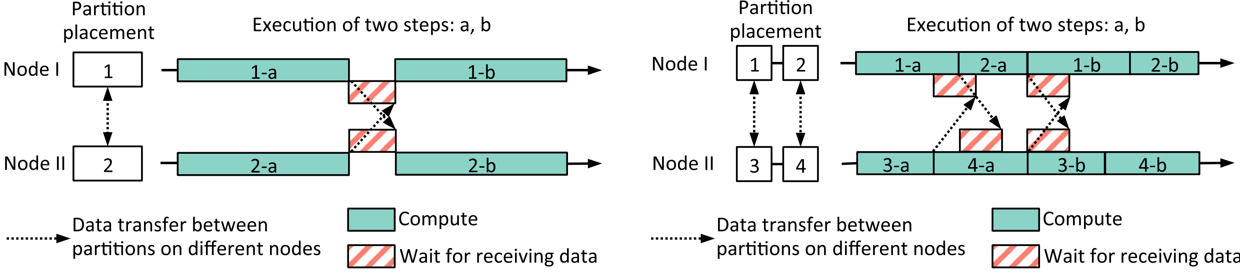
When does dynamic load balancing outperform Birdshot? As further evaluated in Section 7.3, in some cases, Birdshot performs slightly worse than dynamic load balancing algorithms because it cannot make the optimal partition assignment decisions and a poor random partition assignment causes worse performance. However, Birdshot performs better than dynamic load balancing algorithms when the data migration overhead to rebalance load is significant. In such cases, Birdshot balances load without migrating data, achieving better performance.

How does one choose the number of partitions? For the best performance, the number of partitions should be chosen in a suitable range: a sufficient number of partitions to balance load well, while not so many that the overhead to marshal the data transferred between partitions outweighs the benefit. From our experience, using 4 partitions per core is a good rule of thumb point that balances between enough partitions and small overhead unless the load distribution is highly skewed. Many more partitions per core are needed if the simulation is highly sparse, so that the region that has more computation work can be split into enough partitions and distributed to different nodes.

Equation (2) estimates how many partitions are needed based on the percentage of partitions with computation. This value can be experimentally measured. As this percentage changes during simulation, the number of partitions needed changes as well. The maximum of these estimates should be used to ensure good load balancing throughout the entire simulation.

How does cloud performance motivate design decisions? Cloud interconnects must support a wide variety of applications. Furthermore, stragglers, or slow nodes that bottleneck application performance, are common. Stragglers can be caused by workload

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd



(a) With one partition per core, cores are blocked waiting for data communication after completing a step. Partitions *1* and *2* are assigned to two different nodes.

(b) With two partitions per core, data communication is overlapped with computation of the same step or other steps. Partitions *1* and *2* are assigned to the first node, and partitions *3* and *4* are assigned to the second.

Figure 5: *Compare running two simulation steps (a, b) on two single-core nodes with one partition per core or two. Micro-partitioning helps mask communication time.*

imbalance, network congestion or many other problems [AKG*10]. As a result, rather than optimize the network for a particular class of applications, cloud networks use new network designs that provide full bisection bandwidth between all nodes and good worst-case communication performance. [SOA*15]

In modern cloud network architectures [AFLV08, GLL*09, GHJ*09], a rack holds tens of nodes (typically 64). Nodes within a rack and racks themselves are organized in a Clos topology that provides full bisection bandwidth. [SOA*15]. The Clos topology provides multiple paths between two nodes. It is unlikely that all paths are congested, so the two nodes almost always have enough communication capacity to achieve full bisection bandwidth. Amazon claims [web17] that this architecture allows at least 128 C4 instances (1152 cores) with SR-IOV [LTW14] to have full bisection bandwidth, and our measurements have verified that the outgoing throughput of each node is 9.28–9.31 Gbps (averaged over 15 min), when 128 nodes send Iperf [ben17a] TCP test flows to one another.

Second, this interconnect design, combined with the fact that endpoint software dominates network latency, means that the networking performance topology is very flat. That is, the network performance between two nodes on different racks is almost identical to the performance between two nodes on the same rack. It is as if all of the nodes are fully connected. From a networking perspective, each node is equivalent, so there is no need to map the application data layout to the underlying network topology.

Full bisection bandwidth and a flat network performance topology make static random placement an effective scheduling algorithm. Many cloud systems randomly place data on nodes; because their networks are designed to handle such a workload, Birdshot scheduling can follow the same approach.

6. Optimizing Communication Performance

This section describes how Birdshot scheduling optimizes two common communication patterns in distributed fluid simulations. The first pattern is nearest-neighbour communication, i.e. geometrically neighbouring partitions transfer data between each other. The second pattern is barrier operations, such as reduction (e.g. summing integers sent by nodes) and broadcast (e.g. sending an integer to all nodes), which are heavily used in implicit solvers.

6.1. Masking communication time

This subsection gives an example of how Birdshot overlaps communication and computation in nearest-neighbour communication and why micro-partitioning is needed. When each core is assigned one partition, the core alternates between computing one step and waiting for data to continue to the next step as shown in Figure 5(a). Cores are idle at the end of each step.

Figure 5(b) shows how Birdshot masks communication time when each partition in the former setting is split into two, i.e. each core is assigned two partitions on average. In the example, there are four data transfers between the tasks in step *a* and step *b*: from *1-a* to *3-b*, from *3-a* to *1-b*, from *2-a* to *4-b* and from *4-a* to *2-b*. First, communication can happen in the middle of a step and overlap with computation of the same step. For example, once task *1-a* completes, it sends data to task *3-b*. Since tasks in the same step can run in parallel, both task *3-a* and task *4-a* can overlap with the communication time while task *3-b* is waiting for data. Second, communication can overlap with computation of different steps. For example, task *2-a* is executed at the end of step *a*. After the task completes, it sends data to task *4-b*. However, task *3-b* is ready to run while the data are being sent, so the task can overlap with the communication time.

6.2. Optimizing barrier operations

Existing barrier operation implementations trade off between having more hops or having a central message processing bottleneck. This subsection describes a communication library that breaks the trade-off.

Take broadcast as an example. MPI chooses a binomial tree as the broadcast topology for a small message. The topology avoids having any node process too many messages, but the cost is that a message traverses multiple nodes before reaching a leaf node [TRG05]. In

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

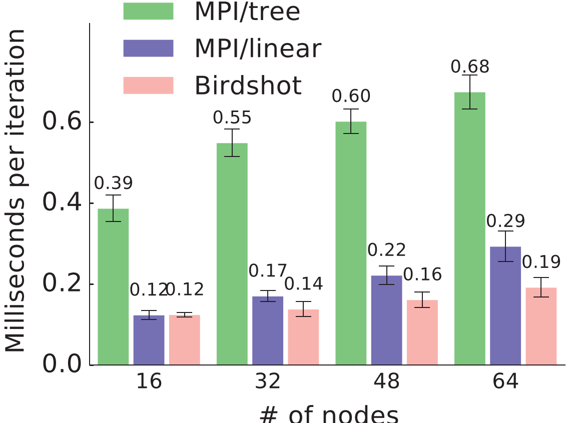


Figure 6: *The iteration times for reduction operations depending on the number of nodes.* `MPI/tree` *uses a binomial tree and is the slowest due to too many communication hops.* `MPI/linear` *uses a linear topology, where a single node exchanges integers with all other nodes, reducing the number of hops but incurring centralized message bottleneck.* `Birdshot` *uses a linear topology on top of a user-level TCP stack for higher message throughput and is the fastest.*

the cloud, it is faster to use a linear topology in which a single node directly exchanges messages with all other nodes. Figure 6 shows how long it takes to reduce the sum of integers and then broadcast the sum back to nodes with different configurations in the cloud. MPI is 2.3–3.3 times faster when using the linear topology instead of the tree topology.

The problem with the linear topology is that a single node sends and receives a large number of network packets, so Birdshot's communication library improves packet throughput using a user-level TCP stack, mTCP [JWJ*14]. mTCP enables a user thread to access the hardware queues of the network interface controller, and avoids context switching between the user space and the kernel space.

Birdshot's communication library enhances mTCP in two aspects. On one hand, Birdshot adds a delayed acknowledgement mechanism to mTCP in order to reduce the number of packets to send. Birdshot delays sending an acknowledgement packet for a fixed time period (e.g. 3× the round trip time). During the time period, if another packet is to be sent, the acknowledgement packet can be piggybacked, so two separate packets can be merged into one. On the other hand, Birdshot batches packet processing to reduce synchronization overhead. First, mTCP delivers received packets to Birdshot's thread in batches. Second, Birdshot enhances mTCP to support queuing packets on different TCP channels within a single context switch.

As shown in Figure 6, Birdshot's communication library runs the barrier operations 30% faster than MPI, even though MPI's barrier operation implementation is highly optimized. Note that the proposed implementation cannot scale infinitely, because the central node in the linear topology, even if well optimized for packet throughput, will eventually become a performance bottleneck beyond thousands of cores.

7. Evaluation

We evaluate four things: (1) how much micro-partitioning and randomized partition assignment improve the end-to-end performance of simulations, (2) how the number of partitions affects performance, (3) how Birdshot performs when using different numbers of nodes and (4) how well Birdshot performs compared with other load balancing algorithms. Birdshot scheduling uses a task-based runtime implemented in C++ [QMSL18]. MPI implementations (Open MPI 1.6.5 [url17]) are used as a reference point without micro-partitioning or randomized assignment. All experiments use Amazon EC2 `C4.8xlarge` nodes in the `us-west-1` region unless specified otherwise (OpenVDB simulations use Google Cloud). A node has two Intel Xeon E5-2666 2.6GHz processors and 60GB memory. The processor has 9 physical cores. Nodes are connected by 10Gbps Ethernet and are allocated in the same 'placement group' that enforces full bisection bandwidth between nodes. The round trip time between nodes is about 100μs.

We report the execution time averaged over iterations (for nested loops, the outer iteration is reported) and cores, and further split the time into CPU *busy* time when a core is running computation codes, and CPU *idle* time when a core is waiting for data from other nodes.

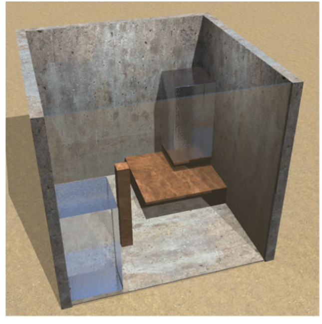
7.1. Benchmarks

Four simulations are used in the evaluation. Each simulation is configured to be as large as possible that will fit in the memory of 8 nodes. These simulations use a variety of data structures, including uniform grids, particles, meshes and sparse grids.

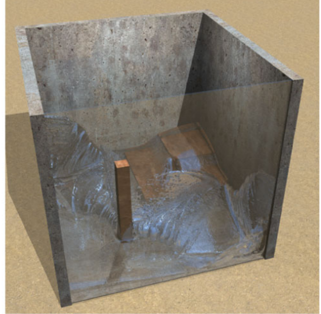
PhysBAM (uniform grids). We use a water simulation from an open-source simulation library, PhysBAM [DHF*]. The simulation uses the particle-level set method with marker particles at both sides of the water interface, and solves incompressibility with a conjugate gradient Poisson solver using an incomplete Cholesky preconditioner. Developing highly scalable Poisson solvers is an active research area [CZY17] and beyond the scope of this paper. The simulation takes a simple approach to distribute every Poisson solver iteration, with multiple inter-node data transfer and synchronization steps per iteration. The preconditioner is split into blocks such that the preconditioning step can run on partitions in parallel. The simulation simulates two sources that pour water into a cubic container split into 432^3 cells. The initial water volume is 20% of the whole domain, and the simulation runs for 600 iterations when the water volume reaches 30%.

PARSEC (particles). The *fluidanimate* benchmark from the PARSEC benchmark suite [Bie11] simulates an incompressible fluid using smoothed-SPH. The simulation stores particles in a Cartesian grid and uses a cut-off distance equal to the length of a cell, such that calculation in one cell only interacts with neighbouring cells. The benchmark simulates 1100×380×1100 cells and 100 iterations. About 25% of cells have particles. There are 600 million particles in total.

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

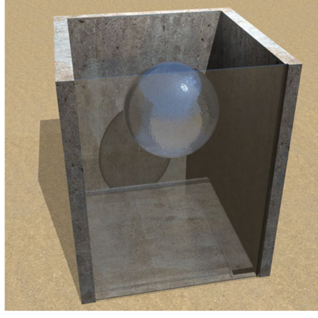


(a) Start scene.

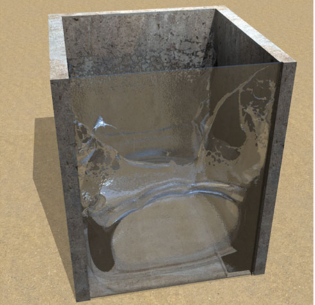


(b) End scene.

Figure 7: *Two-way dam break simulation.*



(a) Start scene.



(b) End scene.

Figure 8: *Sphere drop simulation.*

Lassen (meshes). The Lassen benchmark [ben17b] is a mesh-based simulation modelling how waves propagate from point sources by tracking the wave front using an Eulerian approach. Almost all computation is performed on the cells within a narrow band near the wave front. In each iteration, the simulation calculates how the wave front moves based on the states of the cells in the narrow band, and then updates the position of the narrow band. The mesh is set up as a Cartesian mesh of 5 billion cells, but the application only assumes an unstructured mesh. The benchmark runs 1000 iterations and places multiple point sources on a diagonal plane.

FLIP (sparse grids on OpenVDB). We implement a FLIP simulation application with the same setup used to evaluate the Speculative algorithm [SHQL18]. The simulation workflow is revised for better visual quality. The FLIP simulation uses OpenVDB [Mus13]'s implementation of sparse grids. The FLIP simulation runs over a 1024^3 grid and the simulation domain is split into $16 \times 8 \times 16$ partitions. Eight Google Cloud `n1-highmem-8` nodes are used with eight cores each. Two simulation scenarios are used as shown in Figures 7 and 8.

7.2. Results

End-to-end performance. Figure 9 shows the improvement from micro-partitioning (`micro`) and randomized partition assignment (`random`). Combining both, Birdshot achieves 2.0–3.4× speedup over MPI in the three simulations.

The experiment is run under three settings for each simulation: (1) `reference`, running on MPI, (2) `micro+no random`, running on Birdshot's runtime with 4–16 partitions per core but without randomized assignment, i.e. each node is assigned partitions that are neighbours in the simulation domain and (3) `micro+random`, running on Birdshot's runtime with both micro-partitioning and randomized assignment.

Micro-partitioning in Birdshot scheduling gives 1.3×, 1.5× and 2.0× speedup (from `reference` to `micro+no random`), since micro-partitioning enables masking communication time and balancing load between cores of the same node. Birdshot further achieves 1.5×, 1.5× and 1.7× speedup through randomized assignment that balances load between nodes (from `micro+no random` to `micro+random`).

We also evaluate the execution time of Birdshot's runtime when both micro-partitioning and randomized assignment are disabled (the results are not shown in Figure 9). In that case, Birdshot's runtime is within 10% of MPI. The slowdown is due to the overhead of dynamically scheduling tasks between cores.

Using different number of partitions. Figure 10 shows how the number of partitions affects performance under randomized partition assignment. Using more partitions first improves and then degrades performance. The figure shows the average number of partitions per core, but Birdshot does not bind partitions to cores.

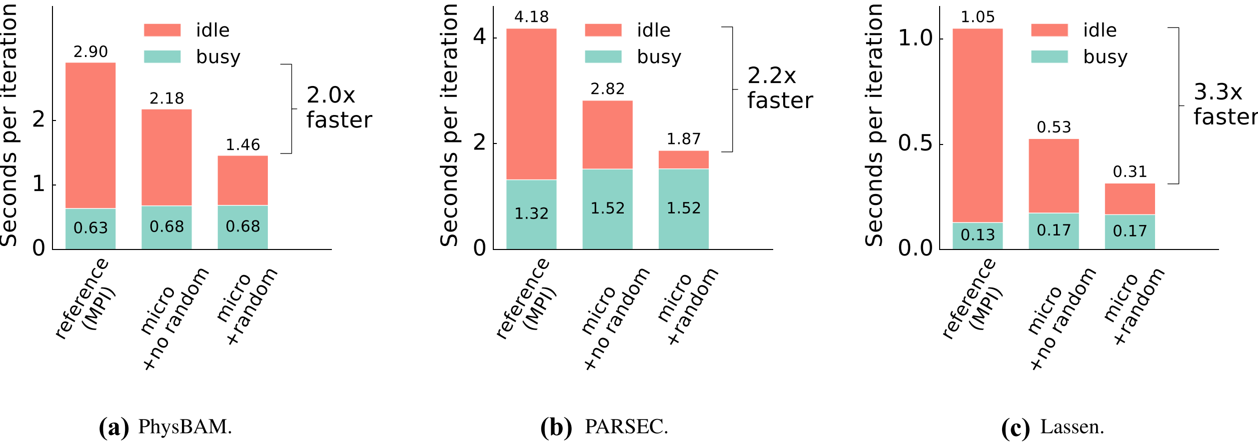
All simulations get the most significant speedup (1.5×, 1.8× and 2.5×) when increasing the number of partitions per core from 1 to 2, because having a 1-to-1 mapping between partitions and cores prevents the runtime from masking communication time or balancing load between cores.

Having too many partitions slows down execution. Firstly, more partitions mean more communication traffic. Secondly, with more partitions, the communication traffic increases, and more computation time is spent marshaling the transferred data.

Simulations achieve the best performance with a medium number of partitions per core. PhysBAM needs between 2 and 4 partitions per core, PARSEC needs between 4 and 8, while Lassen needs between 16 and 32. Lassen needs more partitions to achieve the best performance because its load distribution is more skew.

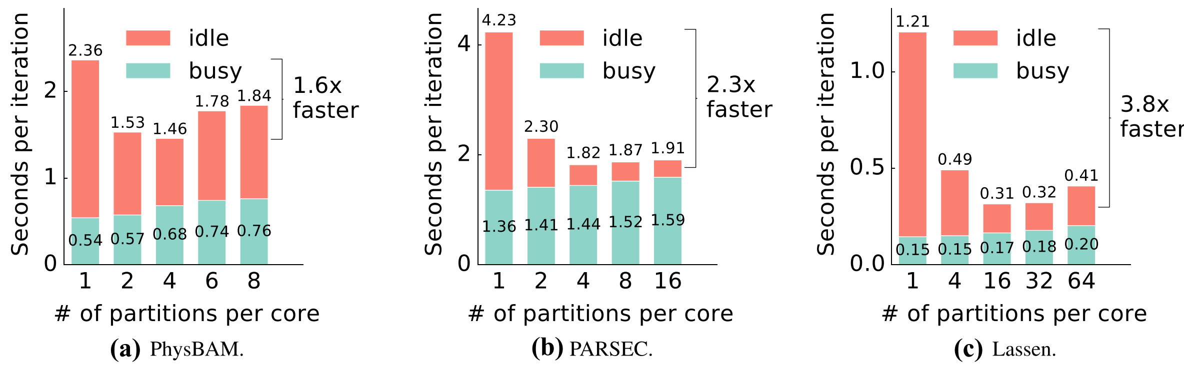
Scalability. We test how Birdshot scheduling scales up to 64 nodes and 1152 cores. Figure 11 shows Birdshot's performance when running PhysBAM on 8–64 nodes while keeping the number of cells per node similar. Birdshot is 1.8–2.1× faster than MPI across different scales. Ideally, the execution time should remain the same as more nodes are used, since both the computation work and the number of nodes increase. However, running a larger simulation gets slower because the computation work of the Poisson solver increases superlinearly with the number of cells. For a larger simulation, the Poisson solver takes more computation iterations to converge, and the computation work of every iteration is linear to the number of cells. Multiplying both, the overall computation work of an outer iteration in Figure 11 increases superlinearly.

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd



(a) PhysBAM. **(b)** PARSEC. **(c)** Lassen.

Figure 9: *Birdshot reduces the iteration times of three simulations by 2.0–3.3× on 32 nodes and 576 cores. The speedup is due to two aspects: using more micro-partitions (* `micro` *) enables masking communication time; assigning them randomly (* `random` *) helps balance load. Note that micro-partitioning slightly increases the computation time due to marshaling transferred data.*



(a) PhysBAM. **(b)** PARSEC. **(c)** Lassen.

Figure 10: *Iteration times depending on the number of partitions on 32 nodes and 576 cores. Too few partitions cannot balance load well. Too many partitions cause execution overhead that outweighs the benefit.*

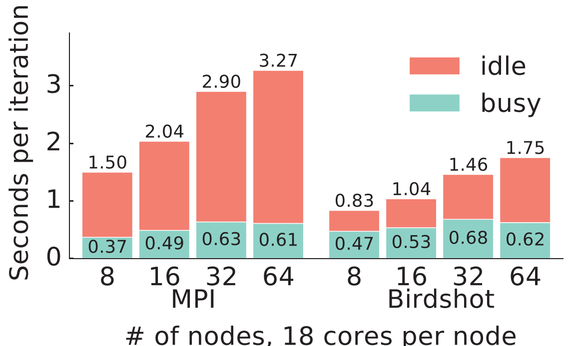


Figure 11: *PhysBAM iteration times when using an increasing number of nodes while keeping the number of cells per node the same. The largest grid size is 528^3 running on 64 nodes and 1152 cores. Birdshot is 1.8–2.1× faster than MPI across different scales. The iteration time increases with more nodes because every shown iteration runs an iterative Poisson solver that converges slower with more nodes.*

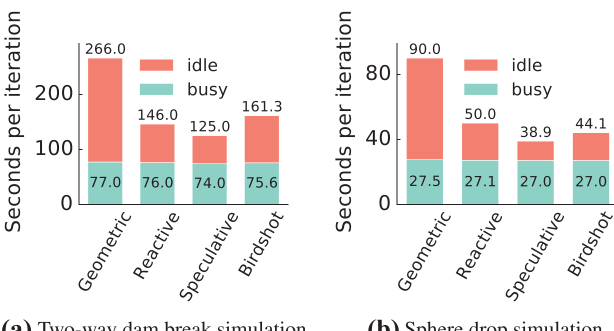
7.3. Comparison with other load balancing algorithms

We compare the performance of Birdshot scheduling with two dynamic load balancing algorithms: *Reactive*, a classic dynamic load balancing algorithm, which periodically migrates partitions between nodes to balance load and *Speculative*, an improved version specialized for fluid simulation [SHQL18] that speculates future load distribution changes to make better migration decisions, which can be seen as a performance upper bound. Both algorithms migrate partitions every 30 simulation time steps.

We ran the exactly same FLIP simulation as in the evaluation of the Speculative algorithm [SHQL18]. The simulation uses sparse grids implemented on OpenVDB. Figure 12 shows the results in two simulation scenarios: a two-way dam break simulation (Figure 7) and a sphere drop simulation (Figure 8). Birdshot scheduling is 1.5× and 2.2× faster than the baseline of static and continuous partition assignment (*Geometric*) because of better load balance.

Birdshot scheduling and *Reactive* have similar performance: *Reactive* is 10% faster in the dam break simulation and Birdshot

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd



(a) Two-way dam break simulation. **(b)** Sphere drop simulation.

Figure 12: *Performance of Birdshot scheduling and three other load balancing algorithms on 8 nodes with 8 cores each. Geometric is a baseline that assigns continuous geometric regions to nodes. Reactive periodically migrates partitions from overloaded nodes to underloaded nodes. Speculative improves upon Reactive by estimating how load distribution changes. Birdshot scheduling is 1.5× and 2.2× faster than the baseline in the two scenarios, achieves comparable performance as Reactive while sometimes outperforming it and is at most 21% slower than Speculative.*

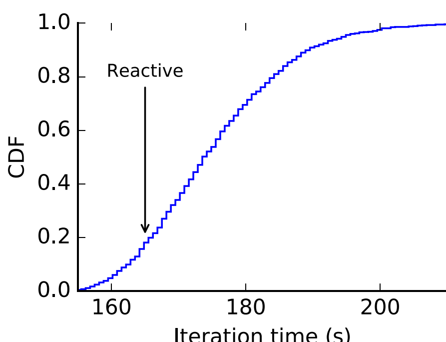


Figure 13: *Distribution of Birdshot scheduling iteration times (for the 100th iteration) in two-way dam break FLIP simulation across 1000 random partition assignments. Reactive takes 165 s in this iteration, but Birdshot takes between 160 and 210 s. So, a poor random assignment may cause Birdshot to perform worse than Reactive.*

scheduling is 12% faster in the sphere drop simulation. Note that the goal of Birdshot is to achieve comparable performance as *Reactive* instead of beating it. Last, Birdshot scheduling performance is within 21% of *Speculative*, which can be seen as a performance upper bound.

We further analyse the root cause of the performance difference between Birdshot and *Reactive*. In the dam break simulation, Birdshot is slower than *Reactive* because of poor random partition assignments. Figure 13 shows the iteration times of Birdshot for one specific iteration under 1000 random partition assignments. Birdshot is faster than *Reactive* with 20% chance but slower with 80% chance. Therefore, a poor random partition assignment makes Birdshot slower.

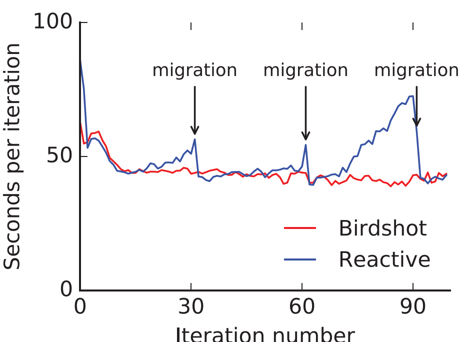


Figure 14: *Comparison of Birdshot and* Reactive *iteration times.* Reactive *migrates data to balance load every 30 iterations.* Reactive *is faster than Birdshot immediately after each migration, but it gradually becomes slower because load distribution changes break balance. So, Birdshot is faster overall.*

In the sphere drop simulation, Birdshot is faster than *Reactive* because it balances load regardless of how dynamic the load is. The sphere drop causes highly dynamic load distribution. Therefore, adjusting partition assignment allows *Reactive* to balance load for one iteration, but later iterations run slower as shown in Figure 14. Birdshot relies on random partition assignment to balance load, running consistently fast regardless of load changes.

8. Conclusion

Over the past 10 years, the computing cloud has grown and evolved to be a platform that powers a wide range of distributed computing applications, but it has been mostly untapped by graphical fluid simulations due to the complexity and the performance problems of distributed computing.

This paper proposes Birdshot scheduling, a simple yet effective solution to accelerate distributed fluid simulations. The key idea is micro-partitioning, which exposes the massive parallelism in fluid simulations to greatly improve the load balance and communication performance. The result demonstrates that the cloud is a promising platform for running graphical fluid simulations, but doing so requires developing new scheduling techniques based on the different workload characteristics and drawing solutions from both cloud computing and scientific computing.

Appendix

This Appendix describes how to compute the expectation of the load imbalance factor in the model proposed by Section 5. In the model, there are n nodes, s partitions per node and p percentage of busy partitions. The number of busy partitions assigned to a node can be seen as the sum of s independent Bernoulli distribution random variables with a parameter of p, which is mathematically equal to a binomial distribution $B(s, p)$. Therefore, the maximum number of busy partitions assigned to a node, denoted as M, is the maximum of n random variables draw from a binomial distribution $B(s, p)$. According to asymptotic theory [NM02], when n is large enough,

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

the accumulative distribution of M is close to:

$$Pr\Big(M \le \sqrt{p(1-p)s}Y_n x + ps + \sqrt{p(1-p)s}Z_n\Big) \rightarrow exp\{-exp(-x)\} = f(x), \quad (A.1)$$

where

$$Y_n = \frac{1}{\sqrt{2\log n}}; Z_n = \sqrt{2\log n} - \frac{\log\log n + \log 4\pi}{2\sqrt{2\log n}}.$$

The probability density function of M can be computed by taking derivatives on both sides of Equation (A.1) on variable x. Then, the expectation of M can be computed through integrating M weighted by its probability density function. The computation is straightforward and the result is:

$$E(M) \rightarrow \sqrt{p(1-p)s}Y_n K + ps + \sqrt{p(1-p)s}Z_n, \quad (A.2)$$

where $K = \int_{-\inf}^{+\inf} xf'(x)dx$. Throwing away high-order terms, we can get:

$$E(M) \approx p \cdot s + p \cdot s\sqrt{2\left(\frac{1}{p} - 1\right)}\sqrt{\frac{\log n}{s}}. \quad (A.3)$$

$LIF = M/(ps) - 1$, so $E(LIF) = E(M)/(ps) - 1$, which gives the Equation in Section 5:

$$E(LIF)|_{p,s,n} \approx \sqrt{2\left(\frac{1}{p} - 1\right)}\sqrt{\frac{\log n}{s}}. \quad (A.4)$$

References

[AFLV08] Al-Fares M., Loukissas A., Vahdat A.: A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM Conference on Data Communication* (2008), ACM, pp. 63–74.

[AGJ*14] Acun B., Gupta A., Jain N., Langer A., Menon H., Mikida E., Ni X., Robson M., Sun Y., Totoni E., Wesolowski L., Kale L.: Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 647–658.

[AKG*10] Ananthanarayanan G., Kandula S., Greenberg A., Stoica I., Lu Y., Saha B., Harris E.: Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 265–278.

[ARK10] Alverson R., Roweth D., Kaplan L.: The Gemini system interconnect. In *IEEE 18th Annual Symposium on High Performance Interconnects* (Aug 2010), pp. 83–87.

[BBAW15] Bailey D., Biddle H., Avramoussis N., Warner M.: Distributing liquids using openvdb. In *ACM SIGGRAPH 2015 Talks* (2015), ACM, p. 44.

[ben17a] Iperf: The network bandwidth measurement tool. https://iperf.fr/. Accessed September 2019.

[ben17b] Lassen benchmark by Lawrence Livermore National Lab. https://codesign.llnl.gov/lassen.php. Accessed September 2019.

[Bie11] Bienia C.: *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[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), IEEE Computer Society Press, pp. 66:1–66:11.

[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 *IEEE International Parallel and Distributed Processing Symposium* (March 2007), pp. 1–11.

[CBM90] Cheung W. H., Black J. P., Manning E.: A framework for distributed debugging. *IEEE Software 7*, 1 (1990), 106–115.

[CZY17] Chu J., Zafar N. B., Yang X.: A Schur complement preconditioner for scalable parallel fluid simulation. *ACM Transactions on Graphics 36*, 5 (July 2017), 163:1–163:11.

[DC*96] Desbrun M., Cani M.-P.: 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 .

[DHF*] Dubey P., Hanrahan P., Fedkiw R., Lentine M., Schroeder C.: PhysBAM: Physically based simulation. In *ACM SIGGRAPH 2011 Courses*, ACM, pp. 10:1–10:22.

[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.

[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.

[FLR98] Frigo M., Leiserson C. E., Randall K. H.: The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1998), ACM, pp. 212–223.

[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.

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

[GHJ*09] Greenberg A., Hamilton J. R., Jain N., Kandula S., Kim C., Lahiri P., Maltz D. A., Patel P., Sengupta S.: VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (2009), SIGCOMM '09, ACM, pp. 51–62.

[GLL*09] Guo C., Lu G., Li D., Wu H., Zhang X., Shi Y., Tian C., Zhang Y., Lu S.: BCube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM Conference on Data Communication* (2009), ACM, pp. 63–74.

[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.

[GMGK84] Garcia-Molina H., Germano F., Kohler W. H.: Debugging a distributed computing system. *IEEE Transactions on Software Engineering, 2* (1984), 210–219.

[GWW*18] Gao M., Wang X., Wu K., Pradhana A., Sifakis E., Yuksel C., Jiang C.: GPU optimization of material point methods. In *SIGGRAPH Asia 2018 Technical Papers* (2018), ACM, p. 254.

[Har62] Harlow F. H.: *The Particle-In-Cell Method for Numerical Solution of Problems in Fluid Dynamics*. Tech. rep., Los Alamos Scientific Laboratory, New Mexico, 1962.

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

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

[JMM*13] Jeannot E., Meneses E., Mercier G., Tessier F., Zheng G.: Communication and topology-aware load balancing in Charm++ with TreeMatch. In *IEEE International Conference on Cluster Computing* (Sept 2013), pp. 1–8.

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

[JWJ*14] Jeong E. Y., Woo S., Jamshed M., Jeong H., Ihm S., Han D., Park K.: mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), USENIX Association, pp. 489–502.

[Kar03] Karypis G.: Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (2003), ACM, p. 56.

[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), ACM, pp. 6:1–6:11.

[KK96] Karypis G., Kumar V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (1996).

[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), IEEE Press, pp. 857–868.

[LMAS16] Liu H., Mitchell N., Aanjaneya M., Sifakis E.: A scalable Schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics (TOG) 35*, 6 (2016), 201.

[LTW14] Lockwood G. K., Tatineni M., Wagner R.: SR-IOV: Performance benefits for virtualized interconnects. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment* (2014), XSEDE '14, ACM, pp. 47:1–47:7.

[MK13] Menon H., Kalé L.: A distributed dynamic load balancer for iterative applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), ACM, pp. 15:1–15:11.

[MSQ*18] Mashayekhi O., Shah C., Qu H., Lim A., Levis P.: Automatically distributing Eulerian and hybrid fluid simulations in the cloud. *ACM Transactions on Graphics 37*, 2 (April 2018), Article 24.

[Mus13] Museth K.: Vdb: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics 32*, 3 (July 2013), 27:1–27:22.

[NH85] Ni L. M., Hwang K.: Optimal load balancing in a multiple processor system with many job classes. *IEEE Transactions on Software Engineering 11*, 5 (May 1985), 491–496.

[NM02] Nadarajah S., Mitov K.: Asymptotics of maxima of discrete random variables. *Extremes 5*, 3 (2002), 287–294.

[OPR*13] Ousterhout K., Panda A., Rosen J., Venkataraman S., Xin R., Ratnasamy S., Shenker S., Stoica I.: The case for tiny tasks in compute clusters. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (2013), USENIX Association, pp. 14–14.

[OWZS13] Ousterhout K., Wendell P., Zaharia M., Stoica I.: Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP '13, ACM, pp. 69–84.

[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 *Proceedings of the 13th European Conference on Computer Systems* (April 2018).

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd

[RMW*16] Reisch J., Marshall S., Wrenninge M., Göktekin T., Hall M., O'Brien M., Johnston J., Rempel J., Lin A.: Simulating rivers in the good dinosaur. In *ACM SIGGRAPH 2016 Talks* (2016), ACM, p. 40.

[SABS14] Setaluri R., Aanjaneya M., Bauer S., Sifakis E.: Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics 33*, 6 (Nov. 2014), 205:1–205:12.

[SHQL18] Shah C., Hyde D., Qu H., Levis P.: Distributing and load balancing sparse fluid simulations. In *Proceedings of the 17th Annual Symposium on Computer Animation* (July 2018).

[SOA*15] Singh A., Ong J., Agarwal A., Anderson G., Armistead A., Bannon R., Boving S., Desai G., Felderman B., Germano P., Kanagala A., Provost J., Simmons J., Tanda E., Wanderer J., Hölzle U., Stuart S., Vahdat A.: Jupiter Rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 183–197.

[Sta99] Stam J.: Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Tchniques* (1999), ACM Press/Addison-Wesley Publishing Co., pp. 121–128.

[TRG05] Thakur R., Rabenseifner R., Gropp W.: Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications 19*, 1 (Feb. 2005), 49–66.

[url17] Open MPI. https://www.open-mpi.org. Accessed September 2019.

[web17] High performance computing on Amazon web services. https://aws.amazon.com/hpc/. Accessed September 2019.

[Whi12] White W. W.: River Running Through It. https://www.cs.siue.edu/~wwhite/SIGGRAPH/SIGGRAPH2012Itinerary.pdf. Accessed September 2019.

[WTYH18] Wu K., Truong N., Yuksel C., Hoetzlein R.: Fast fluid simulations with sparse volumes on the GPU. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 157–167.

[XLD97] Xu C., Lau F. C., Diekmann R.: Decentralized remapping of data parallel applications in distributed memory multiprocessors. *Concurrency - Practice and Experience 9*, 12 (1997), 1351–1376.

[ZBMK11] Zheng G., Bhatelé A., Meneses E., Kalé L. V.: Periodic hierarchical load balancing for large supercomputers. *International Journal of High Performance Computing Applications 25*, 4 (Nov. 2011), 371–385.

© 2019 The Authors. *Computer Graphics Forum* published by Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd