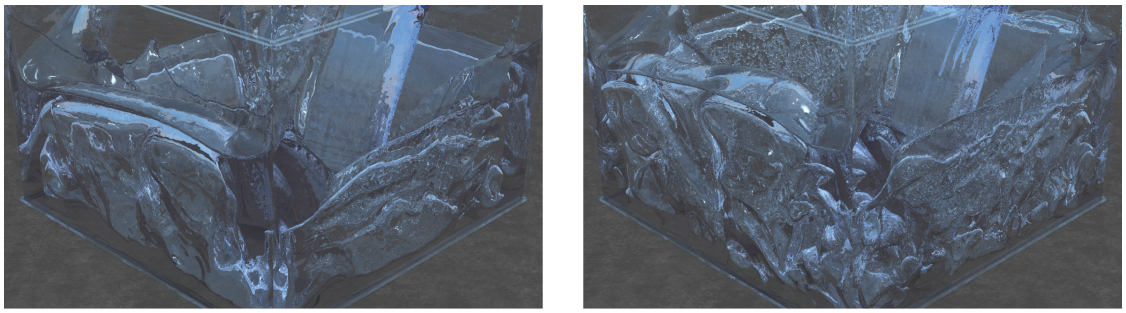
Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud

OMID MASHAYEKHI, CHINMAYEE SHAH, HANG QU, ANDREW LIM, and PHILIP LEVIS,
Stanford University



(a) 152^3 cells, without Nimbus: 335 minutes (b) 256^3 cells, with Nimbus: 268 minutes

Fig. 1. Particle-level set water simulations with and without Nimbus. The left simulation has 152^3 cells, runs on a single core, and takes 335 minutes to simulate 30 frames. The right simulation uses Nimbus to automatically distribute this single-core simulation over eight nodes (64 cores) in Amazon's EC2, simulating faster and with greater detail: 30 frames of a 256^3 cell simulation takes 268 minutes. Without Nimbus, the 256^3 cell simulation takes more than 2 days. Running the 152^3 simulation in Nimbus takes just over 1 hour (68 minutes).

Distributing a simulation across many machines can drastically speed up computations and increase detail. The computing cloud provides tremendous computing resources, but weak service guarantees force programs to manage significant system complexity: nodes, networks, and storage occasionally perform poorly or fail.

We describe Nimbus, a system that automatically distributes grid-based and hybrid simulations across cloud computing nodes. The main simulation loop is sequential code and launches distributed computations across many cores. The simulation on each core runs as if it is stand-alone: Nimbus automatically stitches these simulations into a single, larger one. To do this efficiently, Nimbus introduces a four-layer data model that translates between the contiguous, geometric objects used by simulation libraries and the replicated, fine-grain objects managed by its underlying cloud computing runtime.

Using PhysBAM particle-level set fluid simulations, we demonstrate that Nimbus can run higher detail simulations faster, distribute simulations on up to 512 cores, and run enormous simulations (1024^3 cells). Nimbus automatically manages these distributed simulations, balancing load across nodes and recovering from failures. Implementations of PhysBAM water and smoke simulations as well as an open source heat-diffusion simulation show that Nimbus is general and can support complex simulations.

Nimbus can be downloaded from https://nimbus.stanford.edu.

Authors' addresses: O. Mashayekhi, C. Shah, H. Qu, A. Lim, and P. Levis, Gates Hall, Stanford University, 353 Serra Mall, Stanford, CA 94305; emails: omidm@alumni.stanford.edu, {chshah, quhang, alim16}@stanford.edu, pal@cs.stanford.edu.

ACM 0730-0301/2018/06-ART24
https://doi.org/10.1145/3173551

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

Additional Key Words and Phrases: Cloud computing, Eulerian and hybrid graphical simulations, load balancing, fault recovery

ACM Reference format:
Omid Mashayekhi, Chinmayee Shah, Hang Qu, Andrew Lim, and Philip Levis. 2018. Automatically Distributing Eulerian and Hybrid Fluid Simulations in the Cloud. *ACM Trans. Graph.* 37, 2, Article 24 (June 2018), 14 pages.
https://doi.org/10.1145/3173551

1 INTRODUCTION

Fluid simulation is a cornerstone of modern animation and special effects. These simulations are computationally intensive and so trade off between simulation detail and execution time. Research results often run for several days or weeks, but the turnaround times of production schedules require using lower resolutions.

On-demand cloud computing has made high-performance computing (HPC) clusters immediately available to anyone at very low cost. These ultracheap, highly available computing resources are responsible for the success of real-time "big data" analytics frameworks (Murray et al. 2013; Zaharia et al. 2012) as well as a renaissance in artificial intelligence through deep learning (Dean et al. 2012; Lee et al. 2009).

Despite its benefits, however, cloud computing has been mostly untapped by simulations. Writing high-performance, distributed programs for the cloud is extremely difficult. Cloud computing provides low price points by giving weak service guarantees: nodes, networks, and storage can and do fail quite often (Vishwanath and Nagappan 2010). More perniciously, a small fraction of nodes,

called *stragglers*, perform poorly (Ananthanarayanan et al. 2010), slowing an application to the speed of the slowest core. To perform well in the cloud, an application must interleave computation with communication, recover from failures, and dynamically move work away from slow nodes. For one-off simulations, the up-front engineering costs to distribute in the cloud outweigh the benefits. At the same time, refactoring an existing simulation library is complex and difficult, requiring deep expertise in both distributed systems and simulation methods.

Existing software frameworks provide little help in writing simulations for the cloud. Cloud computing frameworks are designed around application data abstractions such as key-value stores (Murray et al. 2013; Zaharia et al. 2012) or huge graphs (Gonzalez et al. 2012; Malewicz et al. 2010), which do not handle the geometric stencils and solvers used by fluid simulations. At the same time, HPC frameworks leave load balancing and fault tolerance to the programmer (e.g., Charm++ (Kale and Krishnan 1993)), require rewriting simulation kernels to a restricted data model (e.g., X10 (Charles et al. 2005)), or do both (e.g., Legion (Bauer 2014)). Higher-level domain-specific languages such as Regent (Slaughter et al. 2015), Simit (Kjolstad et al. 2016), or Ebb (Bernstein et al. 2016) require rewriting entire simulations from scratch and do not provide the load balancing and fault tolerance that the cloud requires.

This article presents Nimbus, a system that takes an existing grid-based or hybrid (e.g., particle-level set) simulation library and automatically distributes it across many multi-core cloud computing nodes. The core of Nimbus resembles a cloud computing framework: a *driver* program written as a simple sequential program sends execution tasks to a centralized *controller* node that dispatches them to many *worker* nodes. Nimbus's key contribution lies in how it manages data across compute cores so they each seemingly run a stand-alone simulation. Nimbus automatically updates states shared between these sub-simulations (e.g., ghost cells), stitching them together into a single larger one. It achieves this by using a novel, four-layer data model:

(1) In the top layer, a sequential *driver* program executes simulation functions over *geometric* data objects, consisting of a simulation variable over a bounding box.
(2) To efficiently handle ghost regions and other shared subregions, the second layer disjointly subdivides the domain into *logical* objects. Logical objects present an abstraction of a large, shared memory, allowing Nimbus to easily analyze parallelism and data dependencies independently of how the simulation is distributed.
(3) Each logical object can have multiple *physical* instances, residing on one or more nodes. The logical/physical separation allows the controller to abstract away distribution, load balancing, and fault tolerance from the driver program.
(4) Finally, *application* objects are in the format and layout that the simulation library expects. Application objects are typically composed of many physical objects, as they include central as well as ghost regions. Nimbus automatically updates application objects locally or over the network to stitch together many seemingly independent simulations.

To support an existing simulation library, a library developer writes a small number of adapters that translate between library calls and Nimbus's APIs.

Nimbus borrows many ideas from previous cloud and HPC systems, such as a central controller and the distinction between physical and logical objects. Unlike all prior approaches, Nimbus is able to automatically distribute serial simulations using their existing kernels. Furthermore, running graphical simulations in a cloud computing setting exposes new performance bottlenecks that big data applications have yet to encounter. Nimbus therefore introduces new optimization techniques that are crucial for performance, such as caching translations between physical and application objects as well as control plane messages. In summary, this article makes the following contributions:

- A four-layer data model that allows a cloud computing runtime to manage distributed geometric data objects that have overlapping regions such as ghost cells.
- A system architecture that uses the four-layer data model to automatically distribute a simulation across many cores, stitching together many smaller simulations into a single, larger one.
- An evaluation of an implementation of the architecture, called *Nimbus*, quantifying the overheads and scalability of the underlying cloud computing runtime as well as the importance of data caching enabled by using the four-layer data model.
- Demonstrations of PhysBAM (Dubey et al. 2011) particle-level set water and smoke simulations as well as a 3D heat diffusion simulation using a Jacobi solver that Nimbus automatically distributes to run on 1 to 512 cores.

Our evaluation shows that Nimbus runs more detailed simulations faster. By distributing a PhysBAM particle-level set water simulation to run on 64 cores, Nimbus can run a 256^3 simulation faster than a single core can run a 152^3 simulation. Alternatively, Nimbus can lower the 152^3 simulation time from 4 hours to 1 hour. These improvements come without any modifications to PhysBAM libraries, demonstrating the benefits of cleanly separating distribution from simulation methods through Nimbus's system and data model. Furthermore, Nimbus is general; in addition to PhysBAM water and smoke simulations, we demonstrate a 3D heat diffusion simulation (Kamil 2017) distributed by Nimbus.

2 RELATED WORK

This work builds upon and borrows ideas from a diverse body of prior work, including fluid simulation methods, parallel computing, DSLs, and cloud computing.

2.1 Fluid Simulations

There are several common techniques used in fluid simulations. Some, such as smoothed-particle hydrodynamics (SPH) (Desbrun and Gascuel 1996), are purely particle based (Lagrangian). Others are purely grid-based (Eulerian) (Stam 1999). Most modern methods, however, use a combination of particles and grids. The particle-in-cell (PIC) method uses a grid for pressure and viscosity updates but particles for advection (Harlow 1962). The fluid

implicit particle method (FLIP) adds energy and momentum to particles to reduce dissipation (Zhu and Bridson 2005). The affine particle-in-cell (APIC) method augments particles with an affine description of velocity (Jiang et al. 2015). Particle-level set methods use a thin band of particles at the fluid interface to improve accuracy. Enright et al. (2002a), Losasso et al. (2004), and more recent work has explored how to incorporate chimera grids (English et al. 2013) and compressible bubbles (Patkar et al. 2013). PhysBAM is an open-source simulation library (Dubey et al. 2011) that supports several fluid simulations using the particle-level set method. Nimbus is designed to support simulations with an underlying Eulerian structure including particle-level set methods, FLIP, and APIC; it is a poor match for purely Lagrangian or mesh-based simulations.

A data-driven approach to enable real-time fluid dynamics on mobile devices precomputes a large number of state transitions and blends different states into a best-effort simulation (Stanton et al. 2014). Some productions stitch together several coarse and finer simulations, manually fixing the mismatches at simulation boundaries (Reisch et al. 2016; White 2012). These systems could use Nimbus to generate precomputed states or individual simulation frames.

2.2 Parallel Computing

MPI (Snir 1998) is a parallel runtime used in HPC. Extensive developer effort is required for a correct and scalable MPI implementation, such as synchronizing between processes by exchanging messages, and mapping computation workload to processes evenly. Charm++ (Kale and Krishnan 1993) resembles MPI but decomposes processes into smaller object-oriented units called *chares*. Other parallel runtimes such as Legion (Bauer 2014), Sequoia (Fatahalian et al. 2006), and Uintah (Germain et al. 2000) allow developers to describe computations as a sequence of tasks, similar to Nimbus, but require developers to rewrite existing libraries to use their data models. Legion, for example, forces programs to store data in tables and partition those tables along rows or columns; this abstraction works well for Eulerian simulations but performs poorly for particle and hybrid simulations, because adjacent rows in a table can move and become geometrically distant, resulting in highly fragmented partitioning. Sequoia requires programmers to use static arrays, making it difficult to express particle and hybrid simulations. Legion and Sequoia take control of the data model but allow a program to decide how to distribute itself; Nimbus takes the opposite approach, keeping the data model under the control of an existing simulation library and automatically handling distribution. ZPL (Deitz et al. 2004) is an array programming language that provides abstractions to express data parallel computations, and mappings from data to processor sets, and generates MPI or SHMEM code, but requires application code to be rewritten. The array programming model also makes it difficult to express particle or hybrid algorithms.

Task queues with fork-join and work-stealing approaches have been used to parallelize simulation methods for fluid, face, and cloth simulations (Hughes et al. 2007) over multi-core processors. Task queues and work stealing are useful to run parallel tasks over multiple cores on a single node with shared memory but do not scale to multiple nodes.

HPC has a rich literature on load balancing. Static balancing (Catalyurek et al. 2007; Karypis and Kumar 1996) is common for computations on graphs and finite element meshes where each element requires the same number of computational cycles; these approaches pre-compute the optimal partitioning to balance load. Work-stealing (Dinan et al. 2009; Lifflander et al. 2012) algorithms have idle nodes pull computation work from few neighboring nodes, but balance load suboptimally, because each node has only partial knowledge. Load-balancing algorithms based on global load distribution knowledge tend to scale poorly (Ni and Hwang 1985). Hierarchical algorithms (Jeannot et al. 2013; Lifflander et al. 2012; Zheng et al. 2011) scale by decomposing the domain into multiple layers, each of which is tractable to balance. Nimbus takes a centralized approach with a global view of the computation, using a caching layer to avoid centralization from becoming a bottleneck.

2.3 Rendering and Domain-Specific Languages

Languages such as OpenGL and RenderMan are widely researched and used for rendering in the graphics community (Hanrahan and Lawson 1990; The Khronos Group 2017b). Systems such as RenderMan, Chromium, and FlowVR Render support rendering over the network, using multiple nodes (Allard and Raffin 2005; Humphreys et al. 2002). Rendering a large number of frames exploits parallelism, such as temporal parallelism across frames, object parallelism, and even techniques such as sort-first and sort-last, making it a highly parallelizable problem with few dependencies. Languages such as OpenCL and CUDA have made it possible to parallelize computations over a node using thousands of threads (Corporation 2017; Goodnight 2007; Luebke 2008; The Khronos Group 2017a). GPUs make it possible to exploit a large amount of data parallelism, but on a single node.

Halide, a domain-specific language (DSL) for writing image processing pipelines, decouples the algorithm from the computation schedule (Ragan-Kelley et al. 2013). It uses MPI to distribute stencil computations over multiple nodes (Denniston et al. 2016). DSLs such as Liszt (DeVito et al. 2011), Ebb (Bernstein et al. 2016), and Simit (Kjolstad et al. 2016) let programmers express simulation algorithms at a higher level of abstraction suited to the application domain. A programmer writes a sequence of kernels, and the runtime runs these kernels in parallel. These DSLs can run competitively with, or faster than, hand-tuned C with drastically simpler programs. Nimbus is a distributed system that these DSLs can target to automatically distribute and load-balance simulations across many nodes. Regent (Slaughter et al. 2015) exposes logical regions from Legion (Bauer 2014) and automatically generates low-level Legion code. DSLs require that existing libraries be rewritten using a new data model or a new language. With Nimbus, programmers can reuse existing libraries.

2.4 Cloud Computing and Fault Tolerance

Most cloud computing systems, such as Hadoop (Dittrich and Quiané-Ruiz 2012) and Spark (Zaharia et al. 2012), are built upon running map and reduce operations over key-value pairs. These systems assume that all locality is key based: all of the data for a key resides on the same node, but data for different keys has no locality. Correspondingly, these systems support when a task ei-

ther depends on a single partition (a "narrow" dependency) or all partitions (a "wide" dependency) of data, but nothing in-between. This abstraction is poorly suited to the structure and locality of simulation data such as particles and grids. Nimbus extends this task-based approach to support geometric dependencies with data locality.

Cloud nodes can have highly variable performance, due to network cross-traffic, misconfiguration, or hardware faults. This causes a small fraction of nodes to be "stragglers." Cloud systems address stragglers and failures by proactively replicating computations (Ananthanarayanan et al. 2013; Dean and Ghemawat 2008), which works well for disk-bound computations but is prohibitively expensive for CPU-bound ones. Spark tracks lineage for each data set, recomputing and regenerating them if needed.

Naiad uses a timely dataflow to track dependencies and supports richer programs, such as streaming data analysis, machine learning, and graph mining (Murray et al. 2013). TensorFlow automatically distributes machine-learning algorithms, expressed as a dataflow graph of mathematical operations (nodes) and multi-dimensional arrays or tensors (edges) (Abadi et al. 2016). Nimbus borrows the ideas of expressing dependencies using a dataflow graph and using a controller to place data and schedule tasks from these systems. Nimbus provides a richer API to express simulation data, such as particles and grids, and simulation algorithms and solvers.

When the mean number of failures or time between failures is known, approximations such as Young's formula (Di et al. 2013; Young 1974) can be used to compute the optimal number of checkpoints. Systems such as Berkeley Lab Checkpoint/Restart (BLCR) (Hargrove and Duell 2006) and Distributed Multi-Threaded Checkpointing (DMTCP) (Ansel et al. 2009) provide mechanisms and tools to checkpoint processes and restart processes from checkpoints, which libraries and applications can use to recover from faults. However, these systems do not migrate ongoing computations for load-balancing and require users to correctly distribute code, synchronize execution, and exchange messages between processes.

3 FOUR-LAYER DATA MODEL

Nimbus automatically stitches together distributed, sequential, stand-alone simulations into a single, larger simulation. It accomplishes this by using a powerful four-layer data model tailored to the responsibilities of four different program representations, shown in Figure 2. These four layers of abstraction provide a clear separation of concerns for different parts of the system, distributing management of the computation in an efficient manner while hiding the details of parallelism, distribution, and fault recovery from the programmer.

The *geometric* layer is designed to provide a simple abstraction to a simulation author. It is written as a sequential program and completely hides how data is partitioned and computations are distributed. The *logical* layer breaks this geometry into disjoint data objects and is designed to allow the cloud runtime to quickly analyze dependencies as well as how to place and replicate data. The *physical* layer maps the logical objects to specific memory on workers and is designed to allow the distributed worker nodes to replicate objects as well as synchronize them. Finally, the *application* layer is designed to allow existing simulation kernels to use their own custom data structures and data representations without modification.

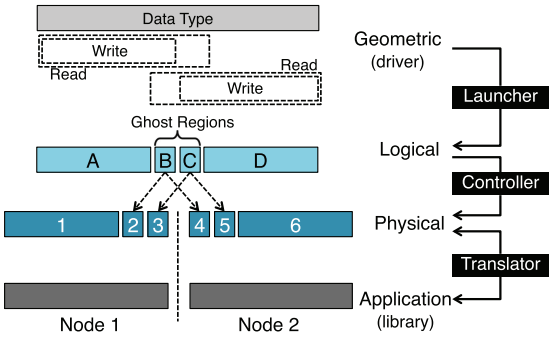


Fig. 2. The Nimbus data model has four abstractions: geometric, logical, physical, and application. This example shows a 1D advection stencil split into two partitions on two different nodes. The geometric view sees the complete simulation domain and applies the stencil to the two partitions. This defines four logical objects, which map to six physical objects on the two nodes. Nimbus assembles these disjoint physical objects into contiguous application objects before invoking simulation library functions.

Figure 2 shows the three software systems that manage the interfaces between these layers. Section 4 discusses these in detail, but the *translator* is notable, because it motivates the need for a four-layer data model. The translator performs the critical conversion between physical objects and the application object formats that a simulation library expects. The translator is critical for performance, because a single application object often consists of many physical objects. For example, node 1's application object in Figure 2 consists of objects 1, 2, and 3. The application expects its object to be a contiguous block of memory to make memory accesses fast, but the worker needs to store it as three separate blocks of memory to make memory and network transfers fast.

The rest of this section explains the four data abstractions in greater detail. Section 4 describes how Nimbus uses each one and translates between them.

3.1 Geometric

A simulation declares a variable as a data type over the simulation domain with a geometric partitioning and a ghost cell width. The simulation library defines the set of available types, such as floating point vectors, particles, or scalars. The simulation loop is a linear sequence of simulation library calls, such as advecting particles or calculating boundary conditions in projection. Each library call specifies its data reads and writes by a set of {variable, bounding box} pairs. The 1D stencil in Figure 2, for example, makes two library calls, one on the left bounding boxes and one on the right:

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

Different simulation variables may have different partitioning. For example, a MAC grid can use a fine-grain partitioning to maximize parallelism while a matrix used during a solve can

be partitioned more coarsely to reduce the number of iterations needed to converge.

3.2 Logical

Using the specified partitioning, Nimbus's launcher automatically translates each variable's geometric domain into a set of disjoint logical objects. Logical objects present the abstraction of a single large, shared memory. They expose how a simulation can be parallelized but hide how the simulation is distributed. Automatically parallelizing a sequential simulation requires carefully managing the read and write order of variables to ensure that a lightly loaded worker does not race ahead and read input data before prior operations on other nodes have completed.

Logical objects define a read/write order to enforce that the parallelized, distributed simulation behaves like the sequential driver program. Each object has a linear sequence of writes, with any number of parallel reads between a pair of writes. This allows Nimbus to analyze data dependencies between calls and enforce execution order. Figure 2 has four logical objects. Nimbus transforms the two library calls into

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

As these two advection calls are in parallel, Nimbus knows that both of them read the same data in B and C and they can execute in parallel. However, the next time this block executes, each advection call reads the output of the prior calls and so cannot run until the prior invocations complete. Nimbus allows in-place operations on regions that have read-write access permissions. However, parallel writes to a region are not allowed. In simulation steps where ghost regions have parallel writes (e.g., particle advection), Nimbus uses a two-stage operation, first creating one temporary object per writer and then reducing them to the final result.

3.3 Physical

Each logical object has one or more associated *physical* objects. A physical object describes actual simulation state stored in the memory of a specific node. Physical objects therefore define how data and computations are distributed. Operations on physical objects describe the simulation's execution as managed by the cloud runtime, including simulation operations and data movement. Nimbus's controller binds references to logical objects to concrete physical instances as it decides how to distribute the simulation.

When distributed across two nodes, the four logical objects in Figure 2 map to six physical objects, three on each node ({1,2,3} and {4,5,6}). The logical library calls above are translated into these physical library calls:

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

and before any future operations that require accessing updated ghost regions, Nimbus synchronizes ghost objects through explicit physical copy calls:

```
network_copy(from:2, to:4);
network_copy(from:5, to:3);
```

Unlike the logical view of objects, which maintain the abstraction of a sequential execution of parallel operators, the physical view describes their actual execution. It presents a level of abstraction that allows workers to intelligently and correctly manage execution while remaining independent of the simulation library they are using.

Physical objects are stored as contiguous blocks of memory, optimized for network transfer and minimizing fragmentation. Section 4.2 describes how Nimbus copies physical objects to ensure that, when it executes, each library call reads the result of the most recent write to an object.

3.4 Application

Nimbus's runtime needs to pass properly formatted data to a library's simulation kernels. When Nimbus invokes a kernel, it does so as if a partition of the larger simulation is itself a stand-alone, smaller simulation. Ghost cells and other replicated subsets of the data are hidden from the library: a single application object typically consists of many physical objects. The simulation library, however, often assumes that the application object is a single, contiguous object in memory. For example, iterating across the application data for Node 1 in Figure 2 requires accessing object 1, then object 2, then object 3.

Nimbus's translator assembles application objects by interleaving data from physical objects. Nimbus uses the geometry metadata of the physical objects to interleave them correctly. Simulation library functions over these application objects can be used unmodified, without rewriting code to use a different data model. Figure 2 shows how the two nodes each assemble a contiguous application-level object out of three disjoint physical objects. For example, a PhysBAM application object can refer to scalar or vector arrays or particles stored as an array of linked lists of buckets of particles.

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

As the call above shows, the simulation library executing on the node is unaware that its application object can be modified by other threads, cores, or nodes. Instead, when a library routine completes, Nimbus automatically updates the underlying physical objects, stitching together the seemingly independent simulations. The controller automatically copies the physical object that contains the latest writes to a logical object to other physical objects. Before Nimbus runs a simulation kernel on application objects, it checks whether they are up to date with their corresponding physical objects. If the physical objects have updates, then Nimbus merges those updates into the application object before running the kernel. For example, when node 2 copies 5 to 3, the next time node 1 runs the kernel it merges the writes to 3 into the corresponding application object. Section 4.3 describes how the translator automatically keeps application layer data objects and their underlying physical objects consistent.

4 SYSTEM DESIGN

This section presents how Nimbus's seemingly complex services can be decomposed into a few simple system components. Nimbus translates between geometric, logical, physical, and application views of the data using the five architecture components shown

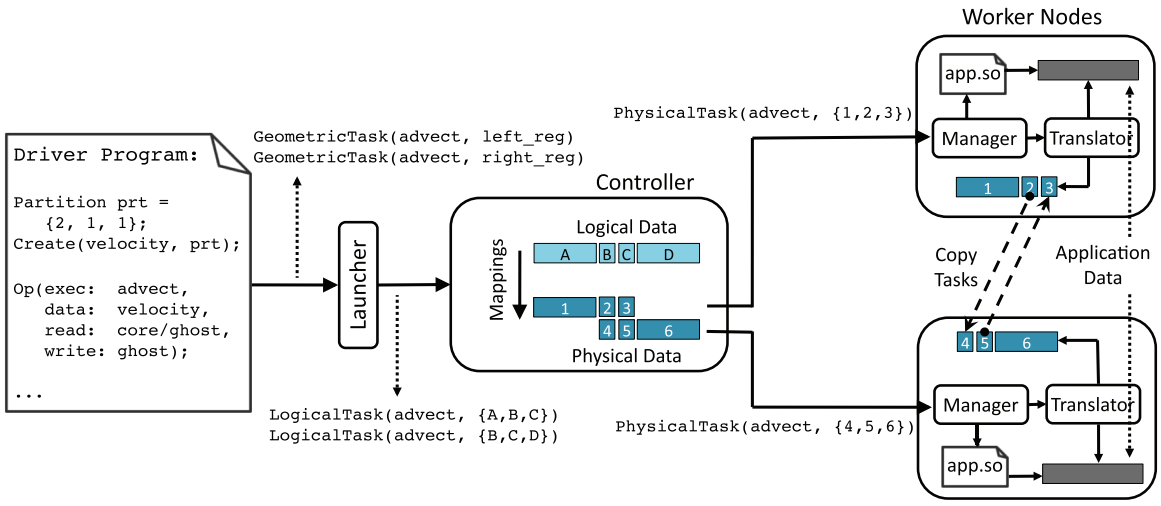


Fig. 3. Design overview of Nimbus. A driver program defines a lineage of operations over geometric data. The launcher turns the driver program into series of parallel tasks over logical data objects and sends them to the controller. The controller assigns tasks to worker nodes for execution, mapping logical objects to specific physical instances. Automatically inserted copy tasks synchronize physical instances when needed. The translator on each computing node assembles application objects out of physical objects, and the manager controls a thread pool to execute library functions over application objects.

in Figure 3. A *driver* program describes the main simulation loop in the geometric view. The *launcher* specifies the logical tasks that each operate on a subset of the simulation domain, defined by logical objects. The launcher sends these logical tasks to a *controller*, which binds the logical objects to physical objects, computes dependencies, and sends task commands to worker nodes. The *manager* running on a worker node receives task execution commands, schedules them based on their dependencies, and invokes simulation library code. The *translator* on worker nodes assembles the disjoint physical objects into application objects and keeps both views consistent with a write-back cache.

4.1 Driver and Launcher

The initialization phase of a driver program specifies the simulation variables, the geometric domain, and a partitioning configuration. Nimbus uses this information to generate a set of logical objects, which decouple data declarations from instantiation, placement, and layout.

The execution phase of a driver program calls library functions over partitions (geometric domains), specifying whether each argument is read-only or read/write. The launcher expands these library calls into waves of *tasks*, assigning one task to each partition. Each task has a *read set* of the logical objects it reads and a *write set* of the logical objects it writes. To compute these sets, the launcher looks up the logical objects associated with each partition. It relaxes the sequential order of the driver program by computing data dependencies between tasks and adding a *before set* to each logical task, which specifies what tasks must complete before this task can safely run. The before set is calculated from read and write sets, enforcing that every task sees the most recent write in the driver's sequential order. These before sets define the *logical task graph*, a DAG that encodes the simulation's potential parallelism.

4.2 Controller

The controller takes the task graph it receives from the launcher and transforms it into a *physical task graph*, which contains execution commands that invoke simulation library functions on specific nodes. Vertices of the physical task graph, like the logical task graph, have a read set, write set, and before set. The read set and write set are physical, not logical, objects, however. A physical object is designed for easy memory management and network transfer: it is a contiguous block of memory that encodes a variable over a domain. Each physical object has a version number.

The controller decides how to distribute the parallel logical tasks from the launcher across worker nodes. To bind logical objects references to particular physical objects, the controller maintains a *context* for each logical task. A context specifies the version number of each logical object the tasks accesses. The controller computes the context by assigning, to each logical object, the maximum version of the contexts of all tasks in the before set. A task that writes to a logical object increments the version number; this newer version propagates to tasks that depend on its result.

A physical task is ready to run when every task in its before set has completed. The controller is responsible for ensuring that when a physical task runs, every physical object it accesses contains the correct version specified by the context in the corresponding logical task. To enforce this invariant, the controller inserts *copy tasks* into the physical task graph as needed. For example, if a ghost region that a task needs to read has been written by a task on another node, the controller inserts a copy task that transfers the data over the network, updating the to-be-read region. In a manner similar to TensorFlow (Abadi et al. 2016), this transforms control dependencies into data dependencies, which has the benefit that nodes implicitly synchronize through their data transfers rather than wait for notifications from the

controller. This transformation ensures that the controller does not become a bottleneck at moderate scales, a problem noted by Sparrow (Ousterhout et al. 2013). Copy tasks can be local or remote. Local copy tasks copy between two memory buffers. Remote copy tasks perform a network transfer using asynchronous I/O. Each computational node has a single I/O thread that handles completion events and updates the physical graph.

When deciding how many physical instances to create, the controller trades off between memory footprint and parallelism with a simple heuristic. It makes multiple instances of ghost regions but keeps a single instance of any central region. For the uniform partitioning of 256^3 simulation in Figure 1, for example, a three-wide ghost region has has 9 to 10,092 cells,[1] whereas a central region has 195,112 cells. Section 6 describes how the controller decides where to place physical objects.

4.2.1 Controller Cache. For large, distributed simulations with hundreds of partitions and millions of logical and physical objects, computing and sending logical tasks to the controller and mapping logical tasks to physical tasks can become a bottleneck. Since simulations involve iterative loops with regular access patterns, they have both regular control flow and dataflow. The launcher and controller therefore cache the tasks they generate, similarly to how execution templates can improve data analytics scalability (Mashayekhi et al. 2017). Cache misses only occur the first time a control path executes (the first iteration, the first time a frame is written to disk, etc.). On subsequent executions, rather than compute and send thousands of tasks, the controller and workers have cached copies of entire task subgraphs, which can be invoked with a single network message; a cache hit allows a single message to schedule tens of thousands of low latency tasks.

4.3 Manager and Translator

The *manager* receives tasks from the controller and manages three task queues: blocked (on an incomplete member of the before set), ready, and running. It maintains a subset of the physical task graph, consisting of the tasks received from the controller. When a task completes, the manager removes it from all task before sets; if this makes a before set of a task empty, that task transitions to the ready queue. The manager maintains a thread pool equal to the number of available cores. When a task completes, it takes the next task of the ready queue and runs it. The ready queue uses two priorities, with each priority having a first in, first out order. Copy tasks have higher priority. This starts asynchronous network transfers as early as possible, interleaving communication and computation.

Before the manager executes a simulation library function, it invokes the *translator* to generate the appropriate application objects. If there is already an application object whose data has the correct versions, then the translator returns immediately. If there are portions of the object that do not contain the most recent write (e.g., a ghost region written to by another node), then it fills into the application object the contents of the physical object specified in the task.

4.3.1 Translator Cache. To prevent unnecessary copies for data that is only used locally, and to remove the need for two copies of central regions, the translator uses a write-back cache. If a task writes to an application object, then the translator does not immediately write out the result to the corresponding physical objects. Instead, it waits until a copy task reads the physical object, at which point it writes the result out to the physical object before the transfer starts. The translator frees the backing memory of physical objects that are out of date with their application object. Because central regions are only transferred when Nimbus load balances between worker nodes, there is only one copy of their data. This allows Nimbus to maintain both the system and application views with only a small (<10%) memory overhead.

[1] The largest ghost region has $3 \times (256/4 - 2 \times 3)^2 = 10,092$ cells.

```
1  class ScalarArray: public AppVar {
2    PhysBAMScalarArray *data ();
3    void Read(DataArray objects, BBox box);
4    void Write(DataArray objects, BBox box);
5    // Internal data members
6    Region bounding_box;
7    PhysBAMScalarArray *data;
8  };
```

Listing 1. Type definition for a float array application object.

5 WRITING SIMULATIONS WITH NIMBUS

Writing a simulation in Nimbus has two parts. First, a simulation library developer writes *adapters* that allow Nimbus to translate between the application and system views and *compute tasks* that encapsulate library function calls. This is a one-time effort. Second, the simulation author writes the driver program that specifies the variables and computational steps of a specific simulation with *control tasks*.

This section explains Nimbus's API, using level set advection as a running example. The application simulates multiple frames, and assumes one iteration per frame for simplicity (PhysBAM dynamically selects a *dt* inversely proportional to the maximum velocity in order to maintain the CFL condition). It explains both the APIs used for adapters as well as a simulation driver.

5.1 Simulation Types (Library Developer)

For Nimbus to translate between the application and system views, a simulation library developer has to write an adapter for the translator to convert them. This adapter is a class that holds the underlying application data. Listing 1 illustrates this API for a scalar array in PhysBAM (e.g., the signed distance). The `Read` method reads out of the application object, translating it into physical objects. The `box` parameter specifies a bounding box subset of the application object to copy. Each element of `objects` also has a bounding box; the data copied is the intersection of the application object bounding box, the physical object bounding box, and `box`. `Write` copies from physical objects into the application object.

5.2 Compute Tasks (Library Developer)

For Nimbus to be able to invoke simulation kernels through tasks, simulation library developers must write adapters between the Nimbus APIs and each kernel. Compute tasks encapsulate calls into simulation library function. They are responsible for setting

```
1  class AdvectLevelset: public ComputeTask {
2    void Execute() {
3      // Get application objects to compute on
4      float& dt = GetAppObject("dt");
5      Vec3fArray& vel = GetAppObject("velocity");
6      FloatArray& sdist =
7        GetAppObject("signed_distance");
8      // Call into the PhysBAM library
9      PhysBAMAdvectLevelset(
10       vel.data(), sdist.data(), dt);
11   }
12 };
```

Listing 2. Compute task for advecting-level set calls into PhysBAM.

```
1  // Define simulation domain
2  BondingBox sim_region = {{0,0,0},{256,256,256}};
3  // Partition the domain along three axes
4  Partitioning partitioning = {2,2,1};
5  // The width of the ghost region
6  int ghost_width = 2;
7  // Center regions with ghost width 0
8  Region center({sim_region, partitioning, 0});
9  // Outer regions includes center and ghost regions
10 Region outer({sim_region, partitioning, ghost_width});
```

Listing 3. Example initialization of a 256^3 simulation domain partitioned in half along the X and Y axes.

any global variables or configuration that the simulation library expects. They also fetch the application objects from the translator that the simulation library function needs. Compute task logic determines when boundary conditions are used (e.g., compute tasks take a different branch if a cell is a boundary). Listing 2 shows simplified code for the `AdvectLevelset` compute task. The compute task reads `velocity` and `signed_distance` over its partition and ghost regions from neighboring partitions, advects `signed_distance` with a call to `PhysBAMAdvectLevelset`, and writes to `signed_distance` in its partition.

5.3 Driver Program (Simulation Author)

The driver program has three parts, shown in Listings 3 through 5. The first part (Listing 3) defines the parameters of the simulation, including the geometric domain, partitioning, and ghost cell widths. This initialization is the one point in the program when a simulation author must consider how to distribute the simulation. Unlike HPC simulations, which tend to perform a uniform computation over data, graphical simulations can have highly varying computations. For example, particle-level set water simulations perform far more computations on water cells than air cells, and water cells on the interface are more computationally intensive than those deep within the volume. As a result, the optimal partitioning depends not only on the type of simulation, but also its initial conditions, and so this is best controlled by the simulation author.

The driver program is written as a control task. Control tasks do not directly invoke simulation functions; instead, they launch other compute and control tasks. A special control task, `Main`

```
1  // Entry point for a simulation
2  class Main: public ControlTask {
3    void Execute() {
4      CreateData("signed_distance", FloatArray, sim_region,
5                 ghost_width, partitioning);
6      CreateData("velocity", Vec3fArray, sim_region,
7                 ghost_width, partitioning);
8      // Create more objects and launch Loop task
9    }
10 };
```

Listing 4. Main task for example particle-level set simulation.

```
1  // A control task to loop until target_frame
2  class Loop: public ControlTask {
3    void Execute() {
4      // Spawn parallel AdvectLevelset tasks
5      LaunchTaskOverAllPartitions(
6        AdvectLevelset,
7        {{"signed_distance", outer},
8         {"velocity", outer}}, // Read outer
9        {{"signed_distance", center}}, // Write center
10       {}); // No task parameter
11     // Spawn other tasks to advect velocity
12     // Spawn next iteration if needed
13     if (parameter.current_frame < parameter.target_frame)
14       LaunchTask(
15         Loop, {}, {} // No read/write data
16         { .current_frame = parameter.current_frame + 1,
17           .target_frame = parameter.target_frame });
18   }
19 };
```

Listing 5. Loop task for example particle-level set simulation.

(Listing 4), is the entry point for simulation. Main defines the simulation variables. These variable definitions create logical objects at the controller. The controller does not create physical objects on worker nodes until it sends tasks to read or write them. This lazy instantiation allows the controller to distribute data only after it has a full picture of task access patterns on data objects.

`Main`, after it launches tasks to initialize the simulation, launches a `Loop` task, which corresponds to the outermost simulation loop. This task launches compute tasks `AdvectLevelset` and `AdvectVelocity` (Listing 5) to compute the next values of `signed_distance` and `velocity`. `Loop` task uses parameters `current_frame` and `target_frame` to determine the end of simulation, or launches another `Loop` control task.

When launching compute tasks, a control task must specify the data that the compute task reads and writes, plus optional task specific parameters. For instance, the `Loop` task specifies read and write sets for `AdvectLevelset` on lines 5 through 10. It does this by specifying the variables and the partitioning to use for reading and writing – an `AdvectLevelset` task over a partition writes to `signed_distance` only over the same partition, by specifying `center` as the partition to write, but reads `signed_distance` and `velocity` from ghost regions over neighboring partitions. Nimbus automatically infers write after read dependencies (e.g., from `AdvectLevelset` to `AdvectVelocity`), and read after write

dependencies (from `AdvectVelocity` to `AdvectLevelset`). Based on these dependencies, it automatically inserts the necessary copy tasks and adds them to before sets, enforcing the correct execution order.

6 FAULT TOLERANCE AND LOAD BALANCING

Production simulations over large grids require hundreds of GB of memory and thousands of hours of CPU cycles. Simulations running in the cloud must deal with straggler nodes, arising from oversubscribed and shared resources such as worker nodes and network capacity, and failures such as I/O failures. Private clusters also exhibit these problems when using a large number of nodes for long periods of time. Disruptions due to failures and slowdown due to straggler nodes can be very expensive in terms of time and effort. Nimbus automatically load-balances applications, and recovers applications from failures (provides fault tolerance).

6.1 Fault Tolerance

The Nimbus controller periodically saves a snapshot of application state and data, allowing it to restart a simulation in case of a failure. These checkpoints are sharded over worker nodes and indexed by a distributed key-value store on top of leveldb (Ghemawat and Dean 2017). At every checkpoint, Nimbus saves the current task graph, which includes all control and compute tasks, and system data objects. Worker nodes periodically send a heartbeat message to the controller. When the controller does not see heartbeat messages from a worker, it assumes that the node has failed. When a node fails, the controller resets the state to the most recent checkpoint, reassigns partitions and tasks across the remaining nodes, and resumes the simulation.

6.2 Load Balancing

Nimbus worker nodes periodically send total time spent in computation tasks to the controller. A high compute time to total time ratio indicates that a node may be a straggler—the node takes more time to complete compute tasks, while other nodes are blocked on it for updated ghost data. Such imbalance can come from oversubscription of shared resources or interference from other applications, difference in CPU speeds, or even from within the application, for instance, due to varying amount of fluid. For instance, a particle-level set fluid simulation may have more computation near the interface, due to additional particles. Most fluid simulations, such as particle-level set, FLIP, and APIC have different amounts of fluid in different partitions, as the simulated fluid evolves over time. This results in variation across partitions and time. A low compute time to total time ratio triggers migration—Nimbus moves some partitions from the straggling worker to neighboring workers. The controller sends commands to migrate tasks and data to worker nodes, and worker nodes exchange data accordingly. This is repeated until the ratio of compute time to total time falls within a threshold. If a worker is particularly slow, then all the tasks from that node may be moved to neighboring nodes.

7 EVALUATION

The goal of Nimbus is to automatically distribute existing sequential fluid simulation libraries to run in the cloud. To evaluate whether it achieves these goals, we ported three graphical simulations to use Nimbus: a 3D heat distribution simulation with a Jacobi iterative solver, a particle-level set water simulation, and a particle-level set smoke simulation. This section presents results on how many lines of code it took to port them, the performance improvements of the distributed versions, and how well the distributed versions scale. It evaluates Nimbus's robustness to stragglers and node failures.

Unless otherwise stated, all experiments use Amazon EC2 compute-optimized instances, since they are the most cost-efficient option for compute-bound workloads. Worker nodes are `c3.2xlarge` instances with eight virtual cores and 15GB of RAM. All nodes are allocated within a single placement group and so have full bisection bandwidth. The Nimbus code repository, including experiments, is publicly available at https://github.com/omidm/nimbus.

7.1 Heat Distribution

We use a 3D heat distribution simulation to measure the overhead that Nimbus introduces independently of any scalability bottlenecks that a simulation method introduces. The 3D heat distribution simulation uses a Jacobi iterative solver and is a common benchmark in prior work (Peraza et al. 2013; Rivera and Tseng 2000; Tang et al. 2011; Williams et al. 2006). Moreover, when the simulation's ghost width is set to 0, the simulation is perfectly parallelizable (there is no shared state between partitions), and the simulation scales linearly.

We took an existing, open-source sequential C++ implementation (Kamil 2017) and ported it to Nimbus by writing 118 lines of code for the translations between physical and application objects and a 94 line driver program. These 212 lines of code allow Nimbus to automatically stitch together many distributed sub-simulations into a single, large simulation.

7.1.1 Scalability. Figure 4 shows the speedup gains from using Nimbus to distribute the 3D heat distribution simulation over 2–64 workers. When the ghost width is 0, each sub-simulation runs independently. With 64 workers, the simulation runs 63.8× faster than on a single worker: Nimbus's overhead is <0.4%. Increasing the ghost width introduces data exchanges between sub-simulations. Because `c3.2x large` instances have only 1Gbps links, these data exchanges become a bottleneck; the simulation method limits scalability. Using a ghost width of three cells, 64 worker nodes speed up the simulation by 8.2×. We verified that communication is the bottleneck by using a distributed logging mechanism similar to block-time analysis (Ousterhout et al. 2015).

7.1.2 Translator and Controller Cache. Nimbus's translator and controller caches are designed to address performance bottlenecks that graphical simulations introduce but traditional cloud systems do not encounter. To quantify the performance benefits of the translator and controller caches, we ran simulations with a ghost cell size of 1, as this is the fastest setting that exercises the translator cache.

Figure 5 shows the results. The translator cache improves performance by 72%. Without a translator cache, Nimbus is forced to translate central regions back and forth between the physical and

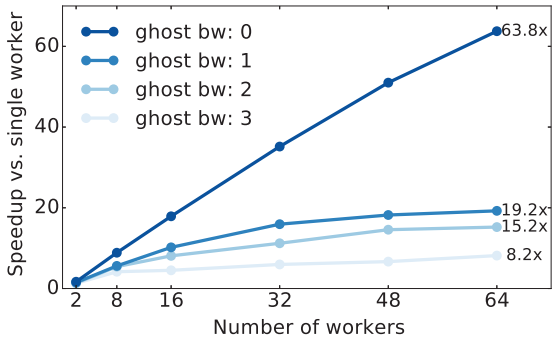


Fig. 4. Heat-3D automatically distributed under Nimbus with different ghost widths. Nimbus scales almost linearly if there is no data exchange among the workers (ghost width 0). However, limited networking throughput in the cloud bounds the speedup gains when there is data exchange among workers. Wider ghost regions increase the networking overhead and lower the gains.

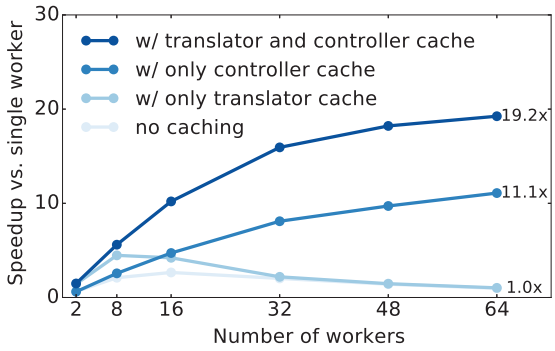
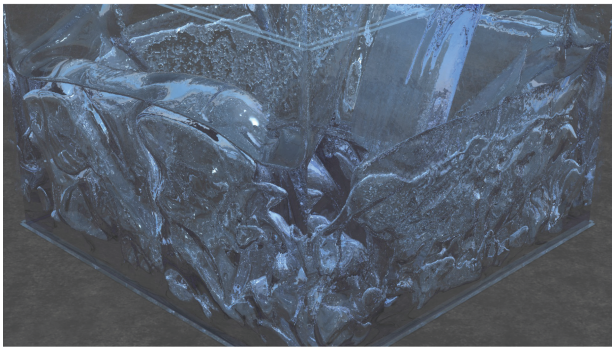


Fig. 5. Effect of translator and controller caches on scalability of the Heat-3D application under Nimbus with a ghost width of 1. The translator and controller caches are crucial for Nimbus's scalability. Without the translator cache, the applications run almost two times slower. Without the controller cache, the controller becomes a bottleneck at even moderate scales and limits the performance gains from scaling.

(a) 128^3, w/o Nimbus: 172 minutes



(b) 256^3, w/ Nimbus: 268 minutes, w/o Nimbus: >48 hours

Fig. 6. Particle-level set water simulations with and without Nimbus. The top simulation has 128^3 cells, runs on a single-core, and takes 172 minutes to simulate 30 frames. The bottom simulation uses Nimbus to automatically distribute this single-core simulation over eight nodes (64 cores) in Amazon's EC2, simulating with greater detail: 30 frames of a 256^3 cell simulation takes 268 minutes. Without Nimbus, the 256^3 cell simulation takes more than 2 days. Running the 128^3 simulation in Nimbus takes only 43 minutes.

application layers after each access. These large memory copies take almost as long as the computations. The controller cache allows Nimbus to scale out to many workers, such that performance is limited by simulation computation and communication. Without a controller cache, the controller becomes a bottleneck as the number of workers increases: although 8 workers see a 4× speedup, adding more workers adds to the control plane load such that 64 workers are no faster than 1.

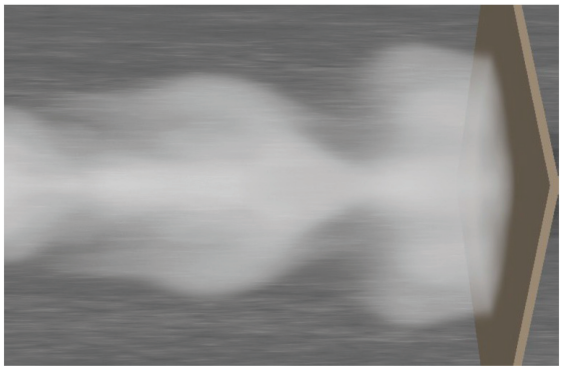
7.2 Particle-Level Set Simulations

To evaluate how well Nimbus can distribute complex, production-quality simulations, we ported two PhysBAM (Dubey et al. 2011) fluid simulations: water and smoke. We use PhysBAM for two reasons. First, it is open source, so one can write translators for its data structures and results can be easily reproduced. Second, PhysBAM is widely used in practice; movie studios such as ILM and Pixar use PhysBAM in production films, and its developers have received two Academy Awards for contributions to special effects (Academy of Motion Picture Arts and Sciences 2017).

We focus on the water simulation, because it is a canonical example, requires an implicit solve, and employs methods that are required for other fluid simulations, such as smoke and fire. We discuss the smoke simulation in Section 7.3. The water simulation has more than 40 different variables in the form of scalar/vector fields as well as particle buckets and 26 different library kernels. The main loop has an outer loop for advecting velocity and particles and an inner loop for solving the Navier-Stokes equations with an iterative pre-conjugate gradient algorithm (Enright et al. 2002b). PhysBAM supports both single-threaded and multi-threaded execution; we used its single-threaded implementations, because using Nimbus to manage threads across cores leads to better load balancing, as Nimbus can dynamically migrate load across cores.

Figure 6 shows the result of running a water simulation with and without Nimbus. Simulating 30 frames of a serial, single-core implementation takes 172 minutes for 128^3 cells, 335 minutes for 152^3 cells, and over 48 hours for 256^3 cells. Distributing the simulation to run on eight nodes (64 cores), 256^3 cells takes 268 minutes,

(a) 128^3, w/o Nimbus: 94 minutes



(b) 256^3, w/ Nimbus: 132 minutes, w/o Nimbus: >30 hours

Fig. 7. Smoke simulations with and without Nimbus. The top simulation has 128^3 cells, runs on a single-core, and takes 94 minutes to simulate 70 frames. The bottom simulation uses Nimbus to automatically distribute this single-core simulation over eight nodes (64 cores) in Amazon's EC2, simulating with greater detail: 70 frames of a 256^3 cell simulation take 132 minutes. Without Nimbus, the 256^3 cell simulation takes more than a day. Running the 128^3 simulation in Nimbus takes only 28 minutes.

slightly faster than the single-core 152^3 simulation and over ten times faster than the single-core 256^3 simulation. Using Nimbus makes much higher-detail simulations faster.

Porting the PhysBAM library required writing translators for three data types: face arrays, scalar arrays, and particles. The translators are less than 1,500 lines of C++ code. This is a one time cost, and other simulations (e.g., smoke) can reuse the translators. The driver program, which launches control tasks, is 620 lines of code.

7.3 Nimbus in Practice

To evaluate the difficulty of writing simulations in Nimbus, we asked an undergraduate summer intern to port a PhysBAM smoke simulation to Nimbus. Writing the driver took less than a month for the student, who had no prior knowledge in graphics or simulation methods. The majority of the time was spent in understanding the original code and determining the required tasks. The water simulation's translations were directly reusable. Figure 7 shows the results of this effort, comparing a 128^3 smoke simulation run on a single core and an automatically distributed 256^3 simulation run on 64 cores; the 256^3 simulation has much greater detail but runs nearly as fast. We expect that developers who understand the simulation methods well will find porting tasks to Nimbus's API fairly straightforward.

7.4 Load Balancing and Fault Tolerance

A key feature of Nimbus is that it automatically monitors execution progress, reacting to stragglers, balancing load, and recovering from failures. To evaluate these capabilities reproducibly, we constructed controlled conditions that would trigger these cases.

Figure 8 shows two different scenarios for running a 256^3-cell water simulation over a cluster of eight nodes, one in which load balancing and fault tolerance are enabled and one in which they are disabled (this mimics the more traditional approach of static parallelization as used in PhysBAM's MPI support). We configured Nimbus to automatically create checkpoints every 30 minutes, as this imposed a very small overhead yet even for larger simulations it is significantly more frequent than the failure rates we have observed, which are most commonly due to disk I/O failures.

After 10 minutes, we launched CPU-bound background processes on one node to cause it to perform poorly and become a straggler. Without load balancing, all of the other workers wait for the straggler, which limits the speed of the simulation. With load balancing enabled, the Nimbus controller quickly migrates computations and the simulation slows only by the degree of lost computational resources.

After 35 minutes, we forced one node to fail and lose all of its in-memory state. With automatic checkpointing enabled, Nimbus successfully rewinds back to a snapshot and resumes computation with the remaining available resources. This is much faster than the current common approach of manually relaunching the simulation when someone detects it has failed.

7.5 Comparison With Other Methods

To evaluate the overhead of Nimbus's centralized controller, which provides fault tolerance and load balancing, we compare Nimbus with the common approach to distribute high-performance graphical simulations, MPI (Snir 1998). Distributing a simulation with MPI requires a complete rewrite of every kernel, as they must integrate the necessary data communication with their computational steps. PhysBAM has hand-tuned MPI implementations for a subset of its kernels. The resulting statically compiled control flow has very little overhead, but is unable to respond to slow nodes, load imbalance, or failures. If a node crashes or is slow, then the programmer must manually recover and restart the simulation. In contrast, Nimbus automatically handles distribution, schedules around poor performance, and recovers from failures.

For medium to large simulations (e.g., 256^3 to 512^3), Nimbus's performance is within 3% to 10% of an MPI implementation. The performance overhead comes from the centralized controller, which must send tasks to workers. The controller cache causes this overhead to grow linearly with the number of workers.

One final question for Nimbus's scalability is how large a simulation it can handle. Figure 9 shows the performance of a 1024^3-cell water simulation distributed over 512 cores (64 nodes), with each core having 1.5GB of simulation state. To the best of the knowl-

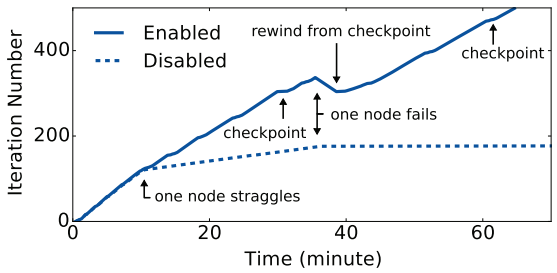


Fig. 8. Running a 256^3-cell PhysBAM water simulation in a cluster of eight Nimbus nodes in two cases: load balancing and fault tolerance enabled and disabled. Nimbus reacts to the straggling node by rebalancing the load and rewinds from the latest checkpoint upon failure. Without these features, the progress speed is bound by the speed of the straggler and any fault halts the simulation.

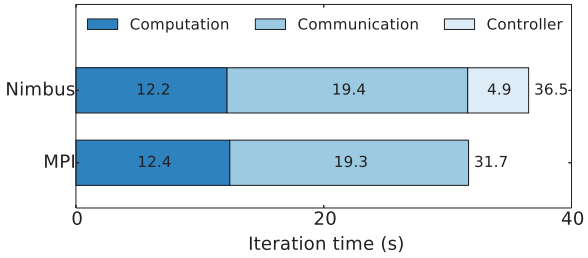


Fig. 9. Running a 1024^3-cell water simulation distributed over 64 nodes (512 cores) under MPI and Nimbus. The data exchange almost exactly matches in both systems. Nimbus's controller is dynamic to be able to react to the straggles and failures; this results in a 15% overhead compared to the MPI implementation with statically compiled control plane, which does not provide load balancing or fault tolerance.

edge of the PhysBAM developers, this is the biggest simulation that PhysBAM has ever been used for; the resulting tiny *dt* values and time for the implicit solve make it impractically slow. Even at such large scales, a single-threaded simulation distributed with Nimbus performs within 15% of a hand-tuned MPI implementation.

8 DISCUSSION AND FUTURE WORK

This article presents Nimbus, a system that distributes grid-based fluid simulations. Nimbus takes a single-threaded simulation and automatically distributes it across many nodes and cores. It achieves this by distinguishing between the application view of data, which is contiguous, and the system view, which subdivides data objects into units corresponding to their data read, write, and transfer requirements. Each simulation node executes as if it is a stand-alone simulation; Nimbus automatically stitches these many smaller simulations into a single, large simulation. Furthermore, it automatically load balances simulations and recovers from failures.

Using PhysBAM particle-level set fluid simulations, we demonstrate that Nimbus can run higher detail simulations faster, distribute simulations on up to 512 cores and run enormous simulations (1024^3 cells). Additionally, Nimbus automatically manages the distributed execution, balancing load across nodes and recovering from failures.

Nimbus is designed to support simulations with an underlying Eulerian structure; this allows it to efficiently map between application and system views of the data. Although it can also support hybrid schemes (e.g., particle-level set, FLIP, APIC), it is a poor match for purely Lagrangian or mesh-based simulations.

Nimbus currently only supports static data resolutions. Adaptive methods provided by data structures such as adaptive meshes (Arney and Flaherty 1990) or hierarchical structures such as VDB (Museth et al. 2013) is left for future work. Supporting these requires that a simulation library can dynamically change and reconfigure the set of logical objects in an efficient way. The multiple resolutions provided by hierarchical data structures can provide valuable information on data placement and load balancing: dividing across nodes on sparse boundaries reduces data movement. Chimera grids (English et al. 2013) are also an area of future work; efficiently supporting them requires abstractions to minimize the data transfer required to synchronize the boundaries of overlapping grids.

Increasingly distributing an incompressible flow simulation causes the solver to consume an increasing portion of iteration time. This suggests that increasingly parallel simulations may benefit from new solvers or centralizing the solver on a single node.

We believe that as simulations continue to be a key aspect of computer graphics and large-scale distributed computing resources continue to become more accessible, Nimbus's approach of auto-distributing simulation codes will grow increasingly important.

ACKNOWLEDGMENTS

First and foremost, we thank Ron Fedkiw and his research group, especially Saket Pakhar, Rahul Sheth, and David Hyde. Over the course of developing Nimbus, they have been tremendously helpful and always available to answer questions about simulation methods and PhysBAM. We also thank Pat Hanrahan and Intel for inviting us to be part of the Intel ISTC-VC and introducing us to the unique systems challenges computer graphics faces. 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 Amazon Web Services Educate program.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

Academy of Motion Picture Arts and Sciences. 2017. Oscar Sci-Tech Awards. Retrieved April 3, 2018, from http://www.oscars.org/sci-tech.

Jérémie Allard and Bruno Raffin. 2005. A shader-based parallel rendering framework. In *Proceedings of IEEE Visualization (VIS'05)*. IEEE, Los Alamitos, CA, 127–134. DOI: http://dx.doi.org/10.1109/VISUAL.2005.1532787

Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. 185–198. http://dl.acm.org/citation.cfm?id=2482626.2482645

Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 265–278. http://dl.acm.org/citation.cfm?id=1924943.1924962

Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. IEEE, Los Alamitos, CA, 1–12. DOI: http://dx.doi.org/10.1109/IPDPS.2009.5161063

David C. Arney and Joseph E. Flaherty. 1990. An adaptive mesh-moving and local refinement method for time-dependent partial differential equations. *ACM Trans. Math. Softw.* 16, 1, 48–71. DOI: http://dx.doi.org/10.1145/77626.77631

Michael Edward Bauer. 2014. *Legion: Programming Distributed Heterogeneous Architectures With Logical Regions*. Ph.D. Dissertation.

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Transactions on Graphics* 35, 2, Article 21, 12 pages. DOI: http://dx.doi.org/10.1145/2892632

Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert Heaphy, and Lee Ann Riesen. 2007. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, Los Alamitos, CA, 1–11. DOI: http://dx.doi.org/10.1109/IPDPS.2007.370258

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices* 40, 519–538.

NVIDIA Corporation. 2017. CUDA C Programming Guide. Retrieved April 3, 2018, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS'12)*. 1223–1231.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1, 107–113.

Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. 2004. Abstractions for dynamic data distribution. In *Proceedings of the 9th International Workshops on High-Level Parallel Programming Models and Supportive Environments*. IEEE, Los Alamitos, CA, 42–51.

Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed Halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM, New York, NY, Article 5, 12 pages. DOI: http://dx.doi.org/10.1145/2851141.2851157

Mathieu Desbrun and Marie-Paule Gascuel. 1996. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Computer Animation and Simulation'96*. Springer, 61–76.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, and others. 2011. 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 (SC'11)*. ACM, New York, NY, 9.

Sheng Di, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, and Franck Cappello. 2013. Optimization of cloud task processing with checkpoint-restart mechanism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'13)*. IEEE, Los Alamitos, CA, 1–12.

J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. 2009. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis (SC'09)*. ACM, New York, NY, Article 53, 11 pages. DOI: http://dx.doi.org/10.1145/1654059.1654113

Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. 2012. Efficient big data processing in Hadoop MapReduce. *Proc. VLDB Endow.* 5, 12, 2014–2015.

Pradeep Dubey, Pat Hanrahan, Ronald Fedkiw, Michael Lentine, and Craig Schroeder. 2011. PhysBAM: Physically based simulation. In *Proceedings of the ACM SIGGRAPH 2011 Courses*. ACM, New York, NY, 10.

R. Elliot English, Linhai Qiu, Yue Yu, and Ronald Fedkiw. 2013. Chimera grids for water simulation. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, New York, NY, 85–94.

Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. 2002a. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys.* 183, 1, 83–116.

Douglas Enright, Stephen Marschner, and Ronald Fedkiw. 2002b. Animation and rendering of complex water surfaces. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'02)*. ACM, New York, NY, 736–744. DOI: http://dx.doi.org/10.1145/566570.566645

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, et al. 2006. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM, New York, NY, 83.

J. Davison de St Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. 2000. Uintah: A massively parallel problem solving environment. In *Proceedings of the 9th International Symposium on High-Performance Distributed Computing*. IEEE, Los Alamitos, CA, 33–41.

Sanjay Ghemawat and Jeff Dean. 2017. LevelDB. Retrieved April 3, 2018, from https://github.com/google/leveldb.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

Nolan Goodnight. 2007. *CUDA/OpenGL Fluid Simulation*. NVIDIA Corporation.

Pat Hanrahan and Jim Lawson. 1990. A language for shading and lighting calculations. *ACM SIGGRAPH Computer Graphics* 24, 289–298.

Paul H. Hargrove and Jason C. Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *J. Phys. Conf. Ser.* 46, 494.

Francis H. Harlow. 1962. *The Particle-in-Cell Method for Numerical Solution of Problems in Fluid Dynamics*. Technical Report. Los Alamos Scientific Laboratory, New Mexico.

Christopher J. Hughes, Radek Grzeszczuk, Eftychios Sifakis, Daehyun Kim, Sanjeev Kumar, Andrew P. Selle, Jatin Chhugani, Matthew Holliman, and Yen-Kuang Chen. 2007. Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors. *ACM SIGARCH Computer Architecture News* 35, 220–231.

Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. 2002. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Trans. Graphics* 21, 3, 693–702.

Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier, and Gengbin Zheng. 2013. Communication and topology-aware load balancing in Charm++ with TreeMatch. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'13)*. 1–8. DOI: http://dx.doi.org/10.1109/CLUSTER.2013.6702666

Chenfanfu Jiang, Craig Schroeder, Andrew Selle, Joseph Teran, and Alexey Stomakhin. 2015. The affine particle-in-cell method. *ACM Trans. Graphics* 34, 4, 51.

Laxmikant V. Kale and Sanjeev Krishnan. 1993. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*. Vol. 28. ACM, New York, NY.

Shoaib Kamil. 2017. StencilProbe: A Microbenchmark for Stencil Applications. Retrieved April 3, 2018, from http://people.csail.mit.edu/skamil/projects/stencilprobe/.

George Karypis and Vipin Kumar. 1996. Parallel multilevel K-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC'96)*. IEEE, Los Alamitos, CA, Article 35. DOI: http://dx.doi.org/10.1145/369028.369103

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, et al. 2016. Simit: A language for physical simulation. *ACM Trans. Graphics* 35, 2, 20.

Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y. Ng. 2009. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML'09)*. ACM, New York, NY, 609–616. DOI: http://dx.doi.org/10.1145/1553374.1553453

Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2012. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*. ACM, New York, NY, 137–148. DOI: http://dx.doi.org/10.1145/2287076.2287103

Frank Losasso, Frédéric Gibou, and Ron Fedkiw. 2004. Simulating water and smoke with an octree data structure. *ACM Trans. Graphics* 23, 457–462.

David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *Proceedings of the 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*. IEEE, Los Alamitos, CA, 836–838.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 135–146.

Omid Mashayekhi, Hang Qu, Chnimayee Shah, and Philip Levis. 2017. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*. 513–526. https://www.usenix.org/conference/atc17/technical-sessions/presentation/mashayekhi.

Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 439–455.

Ken Museth, Jeff Lait, John Johanson, Jeff Budsberg, Ron Henderson, Mihai Alden, Peter Cucka, David Hill, and Andrew Pearce. 2013. OpenVDB: An open-source

data structure and toolkit for high-resolution volumes. In *Proceedings of the ACM SIGGRAPH 2013 Courses*. ACM, New York, NY, 19.

Lionel M. Ni and Kai Hwang. 1985. Optimal load balancing in a multiple processor system with many job classes. *IEEE Trans. Softw. Eng.* 11, 5, 491–496. DOI: http://dx.doi.org/10.1109/TSE.1985.232489

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and VMware ICSI. 2015. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 293–307.

Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 69–84.

Saket Patkar, Mridul Aanjaneya, Dmitriy Karpman, and Ronald Fedkiw. 2013. A hybrid Lagrangian-Eulerian formulation for bubble generation and dynamics. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, New York, NY, 105–114.

Joshua Peraza, Ananta Tiwari, Michael Laurenzano, Laura Carrington, William A. Ward, and Roy Campbell. 2013. Understanding the performance of stencil computations on Intel's Xeon Phi. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'13)*. IEEE, Los Alamitos, CA, 1–5.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6, 519–530.

Jon Reisch, Stephen Marshall, Magnus Wrenninge, Tolga Göktekin, Michael Hall, Michael O'Brien, Jason Johnston, Jordan Rempel, and Andy Lin. 2016. Simulating rivers in the good dinosaur. In *Proceedings of the ACM SIGGRAPH 2016 Talks*. ACM, New York, NY, 40.

Gabriel Rivera and Chau-Wen Tseng. 2000. Tiling optimizations for 3D scientific computations. In *Proceedings of the ACM/IEEE 2000 Conference on Supercomputing*. IEEE, Los Alamitos, CA, 32–32.

Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. 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 (SC'15)*. ACM, New York, NY, 81.

Marc Snir. 1998. *MPI—The Complete Reference: The MPI Core*. Vol. 1. MIT Press, Cambridge, MA.

Jos Stam. 1999. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. ACM, New York, NY, 121–128.

Matt Stanton, Ben Humberston, Brandon Kase, James F. O'Brien, Kayvon Fatahalian, and Adrien Treuille. 2014. Self-refining games using player analytics. *ACM Trans. Graphics* 33, 4, 73.

Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, New York, NY, 117–128.

The Khronos Group. 2017a. OpenCL Overview. Retrieved April 3, 2018, from https://www.khronos.org/opencl/.

The Khronos Group. 2017b. OpenGL. Retrieved April 3, 2018, from https://www.opengl.org/.

Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 193–204. DOI: http://dx.doi.org/10.1145/1807128.1807161

William W. White. 2012. River Running Through It. Retrieved April 3, 2018, from https://www.cs.siue.edu/~wwhite/SIGGRAPH/SIGGRAPH2012Itinerary.pdf.

Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. 2006. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers*. ACM, New York, NY, 9–20.

John W. Young. 1974. A first order approximation to the optimum checkpoint interval. *Commun. ACM* 17, 9, 530–531.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'16)*. 2.

Gengbin Zheng, Abhinav Bhatelé, Esteban Meneses, and Laxmikant V. Kalé. 2011. Periodic hierarchical load balancing for large supercomputers. *Int. J. High Perform. Comput. Appl.* 25, 4, 371–385. DOI: http://dx.doi.org/10.1177/1094342010394383

Yongning Zhu and Robert Bridson. 2005. Animating sand as a fluid. *ACM Trans. Graphics* 24, 965–972.

Received July 2017; revised January 2018; accepted February 2018