A Sparse Distributed Gigascale Resolution Material Point Method

YUXING QIU, University of California, Los Angeles
SAMUEL TEMPLE REEVE, Oak Ridge National Laboratory
MINCHEN LI, University of California, Los Angeles & Timestep Technologies
YIN YANG, University of Utah & Timestep Technologies
STUART RYAN SLATTERY, Oak Ridge National Laboratory
CHENFANFU JIANG, University of California, Los Angeles & Timestep Technologies

In this article, we present a four-layer distributed simulation system and its adaptation to the Material Point Method (MPM). The system is built upon a performance portable C++ programming model targeting major High-Performance-Computing (HPC) platforms. A key ingredient of our system is a hierarchical block-tile-cell sparse grid data structure that is distributable to an arbitrary number of Message Passing Interface (MPI) ranks. We additionally propose strategies for efficient dynamic load balance optimization to maximize the efficiency of MPI tasks. Our simulation pipeline can easily switch among backend programming models, including OpenMP and CUDA, and can be effortlessly dispatched onto supercomputers and the cloud. Finally, we construct benchmark experiments and ablation studies on supercomputers and consumer workstations in a local network to evaluate the scalability and load balancing criteria. We demonstrate massively parallel, highly scalable, and gigascale resolution MPM simulations of up to 1.01 billion particles for less than 323.25 seconds per frame with 8 OpenSSH-connected workstations.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms;**

Additional Key Words and Phrases: Material Point Method, High Performance Computing, distributed system and computing

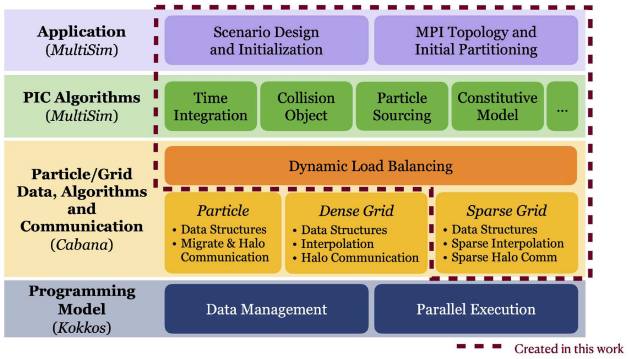

Fig. 1. **Hierarchical System Architecture.** Our distributed MPM simulation system is designed and implemented hierarchically. In this article, we will use *Particle/Grid data layer* in short for the *Particle/Grid Data, Algorithms and Communication layer*. Different layers are implemented in separated codebases. Library names are labeled below layer names.

ACM Reference format:
Yuxing Qiu, Samuel Temple Reeve, Minchen Li, Yin Yang, Stuart Ryan Slattery, and Chenfanfu Jiang. 2023. A Sparse Distributed Gigascale Resolution Material Point Method. *ACM Trans. Graph.* 42, 2, Article 22 (January 2023), 21 pages.
https://doi.org/10.1145/3570160

1 INTRODUCTION

High-resolution simulations are of high demand in both the VFX industry and scientific research. In recent years, the **Material Point Method** (**MPM**), due to its flexibility and versatility, has shown a great potential for modeling a wide range of continuum materials.

To reduce computational cost and programming efforts, researchers have explored modern computational platforms and improved MPM in both parallelization schemes and latent particle/grid data management. Dedicated code design examples in graphics include threaded CPU MPM [Fang et al. 2018], single-GPU MPM [Gao et al. 2018; Hu et al. 2019, 2021], and multiple-GPU implementations [Fei et al. 2021; Wang et al. 2020]. These state-of-the-art solvers still focus on exploiting a single machine with limited memory and computing power, leading to restrictions from several perspectives. On one hand, CPU-based computation is less efficient due to the limited number of threads despite the hundred-GB memory to support large-scale data. GPU-based computation, on the other hand, can significantly reduce the simulation time, but the onboard memory makes it challenging to go large-scale. While

This work has been supported in part by NSF CAREER 2153851, CCF2153863, ECCS-2023780, DOE ORNL contract 4000171342, NSF 2244651 and 2301040. Additionally, this work was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. DOE Office of Science and the NNSA. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.
Authors' addresses: Y. Qiu, M. Li, and C. Jiang, 603 Charles E Young Dr E, UCLA Slichter Hall 3860, Los Angeles, CA 90095; emails: yuxqiu@gmail.com, yxqiu@g.ucla.edu, minchernl@gmail.com, cffjiang@math.ucla.edu; S. T. Reeve and S. R. Slattery, Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, TN 37830; emails: {reevest, slatterysr}@ornl.gov; Y. Yang, 201 Presidents' Cir, The University of Utah, MEB 3454; Salt Lake City, UT 84112; emails: yin.yang@utah.edu, yangzzzy@gmail.com.

0730-0301/2023/01-ART22 $15.00
https://doi.org/10.1145/3570160

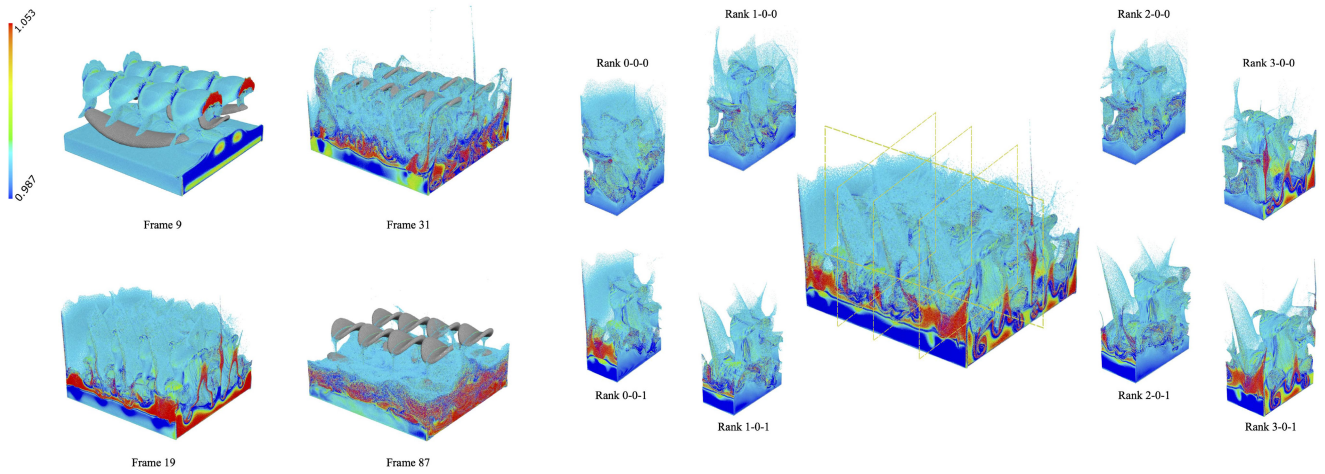

Fig. 2. **1B-Fluid** with more than **1.01B** particles. We use $4 \times 1 \times 2$ MPI ranks (eight in total) to handle the simulation domain. The weakly compressible fluid particles are colored by the volume change ratio. We show representative frames (left) and sub-domains handled by eight workers (right).

using additional GPUs can relieve the intense memory usage [Fei et al. 2021; Wang et al. 2020], the number of GPUs that a single motherboard can hold is still capped. Furthermore, both CPU and GPU MPM require skillful programming dedicated design efforts.

Together, these restrictions motivate our exploration of a device-portable distributed simulation system, which allows researchers with minimal software experience to customize large-scale simulations and maximally leverage their devices. Specifically, we aim to build a distributed MPM system that pursues the following design goals:

- **Device portability for high performance.** Many existing simulations using only CPU or GPU resources require dedicated design, implementation, and optimization of code. It is also challenging to perform device-related code migration if new needs arise. Our design goal is to support effortless hardware switching according to users' needs, *i.e.*, to allow switching the latent programming models and parallel platforms for the simulations by modifying very few lines of code.
- **Distributed dispatch for large-scale simulation.** We allow the simulation system to scale up according to available hardware. To achieve this goal, we need to establish reliable and efficient grid/particle data structures and build communication machinery among multiple separate-memory computing nodes. To further improve the scalability, we aim to reduce unnecessary memory usage by developing new sparse data structures.
- **Dynamic workload decomposition.** Distributing computations to multiple workers is challenging from two standpoints. First, from the performance perspective, calculation time is bounded by the device with the highest workload. While other nodes are busy, idle nodes with tasks completed earlier simply wait, wasting time and resources. Second, robustness and system stability are crucial. An imbalanced partitioning strategy may cause run-time failure by exhausting the memory of some overloaded node. In simulations, the topology of the activated grids and the particles can dramatically differ from their initial settings. Thus, static partitioning can become extremely ineffective and non-robust, working only for carefully designed scenes as in Fei et al. [2021] and Wang et al. [2020]. Therefore, we demand dynamic workload partitioning for better distributed performance and robustness.
- **Programming simplicity.** A typical parallel simulation code requires great programming effort in memory management and parallel execution. We prefer the system's users with different simulation and programming skill levels can all focus on their primary goals, ranging from setting up scenes and designing numerical algorithms to exploring novel data structures.

1.1 Key Insight

These assumptions and design goals lead us to a hierarchical architectural design principle as shown in Figure 1. We divide the whole system into four layers: the *programming model layer*, the *particle/-grid data layer*, the *PIC algorithm layer*, and the *application layer*. This hierarchical design allows users with various experimental goals to focus on distinct layers and to extrapolate the system's potential. Below we discuss each layer in more details.

Programming Model. This bottom layer focuses on developing device-portable specializations on (1) memory allocation and access, and (2) parallel execution operations. Specifically, it allows upper layers to use unified interfaces to perform parallel computations with the desired backend computational models (*e.g.*, OpenMP or CUDA) and manipulate data stored on user-preferred computing devices (*e.g.*, CPU or GPU).

Particle/Grid Data. Generally, Lagrangian particles, Eulerian grids, and/or their combinations are used for simulation schemes considered in this work. However, designing and implementing these data structures and related algorithms on distributed systems require intensive efforts. Thus, we use an independent layer to implement particle/grid distributed data structures that

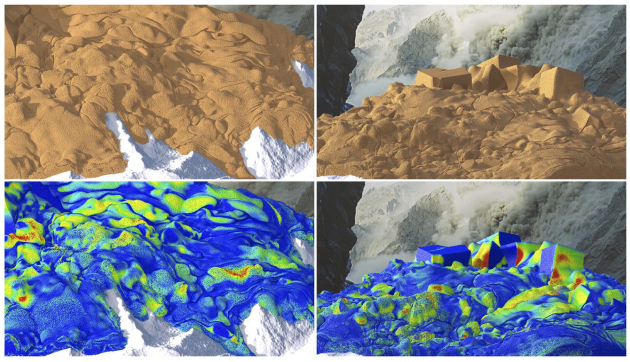

Fig. 3. **Mudflow.** Our distributed MPM enables this over-207.8M-particle mudflow simulation using averagely only 159.34 seconds per frame. Here, we employ four workstations connected through OpenSSH.

allow users to customize the latent memory layout and the attributes stored for each element. Furthermore, particle and grid inter-rank communications are integrated for distributed systems. In addition, since particle/grid number determines the total workload on each rank, we also attach dynamic load balancing as another crucial component in this layer.

PIC Algorithm. Simulating dynamic physical systems typically requires a time integration scheme. In our case, for example, MPM adopts a **Particle-In-Cell** (**PIC**) paradigm. Users can switch to other schemes by modifying the integration strategy. Additional components in this layer include constitutive models for material versatility and a particle sourcing module for time-dependent particle injection.

Application. Users can customize the scene setup and material parameters inside this layer based on all lower-layer components. The user also chooses an MPI topology according to the host hardware.

1.2 Background

To avoid reinventing the wheel, we employ two libraries, Kokkos [Edwards et al. 2014; Trott et al. 2022] and Cabana [Mniszewski et al. 2021; Slattery et al. 2022] to satisfy part of the requirements of the bottom two layers.

Kokkos provides support for basic data structures on all major heterogeneous and high-performance computing architectures [Edwards et al. 2014; Trott et al. 2022]. Users can allocate multidimensional arrays on different computing devices such as CPUs and GPUs in a relatively easy and unified manner. In addition, Kokkos contains abstractions for most general parallel execution patterns that are portable across hardware. Kokkos fully satisfies the design goal of our *programming model layer*, enabling adoptions on modern hardware including NVIDIA GPUs and multi-core CPUs which are both used in this work.

Cabana is a particle-specific library based on Kokkos. It provides particle data structures, particle algorithms, and MPI communication operations. It also supports dense-grid and dense-particle-grid operations with a static partitioning. Thus, Cabana satisfies the particle-related requirements in our *particle/grid data layer*; however, we require extra components for the grid components. First, the dense grid is not suitable for simulations with significant empty space since a large amount of memory and resources would be wasted. Second, static partitioning limits the performance and scalability as analyzed in the design goals.

1.3 Contributions

Following the hierarchical approach above, we develop a distributed simulation framework specialized for MPM kernels, emphasizing scalability and performance portability. Our system is built on top of a modern C++ programming model (Kokkos) and allows users to write and dispatch performant code on HPC platforms with CPU- and GPU-based parallelization. In order to support the generality for users to switch back-end devices effortlessly, we do not pursue extensive performance improvement on problems that can be well-solved by dedicated-designed single-rank CPU or GPU devices, as did in Fei et al. [2021], Gao et al. [2018], Klár et al. [2017], and Wang et al. [2020]. Instead, we concentrate on properly resolving large-scale scenarios where inter-communication is unavoidable and single-rank machines are unable to handle.

In addition, for the *particle/grid data layer*, we utilize Cabana for particle-related operations. We extend the Cabana library by designing and implementing a novel distributed sparse grid data structure with highly efficient allocation, access, and communication algorithms. Furthermore, we customize a dynamic load balancing partitioner to improve the simulation performance by ensuring a balanced workload distribution on all MPI ranks. Based on these implementations, we develop a fully open-source simulation library that supports multiple MPM-related algorithms and application designs, leading to gigascale resolution simulations for a wide range of solid and fluid materials. We further provide comprehensive computational experiments that demonstrate

- the scalability of the proposed distributed system,
- the benefits of dynamic load balancing on sparse simulations, and
- the performance variance with different MPI topologies.

In addition, we demonstrate large-scale simulation examples for designers to customize their scenes with versatile application-level components.

1.4 Overview

Following the hierarchy proposed in Section 1.1, we introduce each system layer in the paper's main body. First, in Section 3, we overview the background needed to understand our article and corresponding implementations, including an introduction of MPM (Section 3.1), the Kokkos programming model (Section 3.2), and Cabana particle-related implementations (Section 3.3). This section thus covers the *programming model layer* and part of the *particle/grid data layer*. Next, we introduce two new features we integrated into the *particle/grid data layer*: Section 4 presents the proposed MPI-dedicated distributed sparse grid and Section 5 shows our distributed dynamic load balancing scheme. After that, we describe additional details related to the *PIC algorithm* and *application* layers in Section 6. We then offer performance analysis in Section 7:

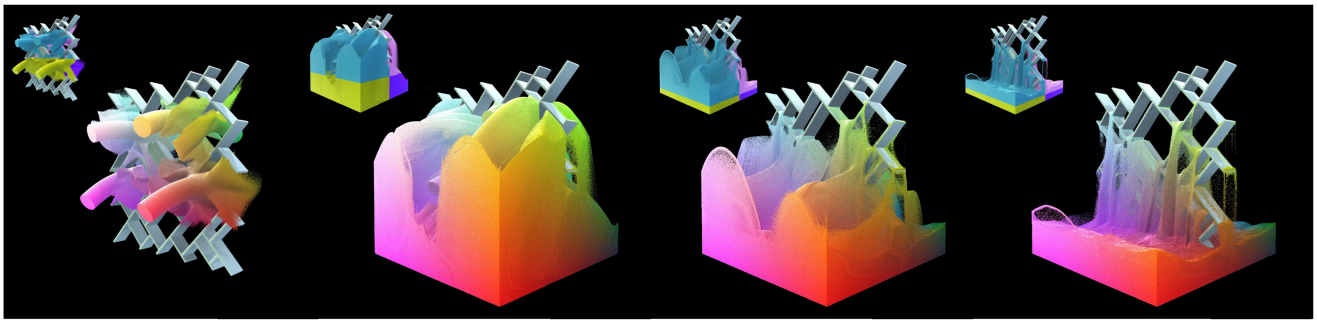

Fig. 4. **High-resolution Sand Injection** with grid resolution $512 \times 512 \times 512$ and 266.5M particles. Sand particles are rainbow-colored by their positions. We use four MPI ranks ($2 \times 2 \times 1$) for computation and show the partition status on the left-top corner of each sub-figure.

(1) weak and (2) strong scaling of the proposed distributed MPM scheme, (3) the performance improvement with our distributed dynamic load balancing algorithm, (4) performance comparison with different MPI rank topologies, and (5) large-scale simulation results with up to 1B particles. Finally, we conclude this article with limitations, discussion, and possible future work.

2 RELATED WORK

2.1 HPC-Oriented Simulation Programming Model

Modern hardware makes it possible to improve simulation performance with dedicated data structure and parallel kernels. One primary attempt is to use multiple CPU cores with tools like OpenMP [Dagum and Menon 1998] and Intel TBB [Willhalm and Popovici 2008]. Further explorations are built upon GPUs for faster computations. For example, GPU-based schemes were designed for Eulerian and Lagrangian fluids [Amada et al. 2004; Chentanez and Müller 2011, 2013; Cohen et al. 2010; Goswami et al. 2010; Pfaff et al. 2010; Vantzos et al. 2018; Winchenbach et al. 2016], as well as for hybrid solvers [Chentanez et al. 2015; Gao et al. 2018; Hu et al. 2019, 2021; Wu et al. 2018]. Multi-GPU platforms [Fei et al. 2021; Wang et al. 2020] were developed for MPM as well.

For scalability, researchers also explored distributed simulations [Bauer et al. 2012; Liu et al. 2016; Qu et al. 2020; Shah et al. 2018]. Kale and Krishnan [1993] introduced Charm++, an object-oriented portable C++-based parallel programming language that is still being actively maintained by researchers from multiple fields. Additionally, supportive systems such as Canary [Qu et al. 2018] and Nimbus [Mashayekhi et al. 2017, 2018] distribute tasks onto computing nodes. For most systems, MPI [Snir et al. 1998] is adopted as the message communication library. It provides various communication primitives for sending and receiving data among ranks. For example, Lesser et al. [2022] proposes a multi-physics framework named *Loki*, which can be used as a generalized tool to simulate various material phenomena ranging from elastic solids to fluids with multi-CPU-core clusters. However, *Loki* leaves GPU usage as future work. Similarly, for other systems, whether single-machine-based or distributed, data arrangement and computations are limited to specific back-end devices, and the performance optimization is only architecture-oriented.

There have been many efforts for performance portability, *i.e.*, enabling high performance across different architectures with a single source code. For example, Hu et al. [2019] developed a compiler that allows users to switch CPU/GPU backend by changing a single line of code. Medina et al. [2014] provided a unified API for interacting with backend devices with a C-extended kernel language. Additionally, Zenker et al. [2016] implemented an abstract hierarchical redundant parallelism model that supports applications on many hardware types ranging from multi-core CPUs to GPUs. Furthermore, libraries such as Kokkos [Edwards et al. 2014; Trott et al. 2022] with its extensions like Cabana [Mniszewski et al. 2021; Slattery et al. 2022] support the manipulation of array-based data structures and their corresponding parallel patterns on multiple underlying computing devices in a distributed manner. These libraries relieve researchers' effort in backend-oriented maintenance and their usage is becoming a trend for next generation high-performance simulations.

2.2 Sparse Grid Data Structures

In many Eulerian and hybrid simulations, the grids are sparsely activated, *i.e.*, only part of the grids contains non-zero entries. Thus, sparse grid data structures have been developed to improve memory bandwidth and data access efficiency. For instance, OpenVDB [Museth 2013], sparse paged grids [Setaluri et al. 2014], and Bifrost's volume tools [Bojsen-Hansen et al. 2021] enable efficient interactions of time-varying sparse quantities over large grid with dedicated grid representation design on CPUs. Furthermore, Hoetzlein [2016], Museth [2021], and Gao et al. [2018] broaden the sparsity idea of VDB and sparse paged grids to GPUs. These extensions vastly improve the simulation scalability and efficiency on NVIDIA GPUs with limit-sized RAMs. Moreover, developing a data hierarchy is an important addition to improve sparse data access efficiency, such as in Hu et al. [2019] and Liu et al. [2018].

2.3 Load Balancing for Simulations

Load balancing and workload distribution are crucial for the performance of distributed systems. Traditional load balancing algorithms perform either geometric-based [Berger and Bokhari 1987] or graph-based [Catalyurek et al. 2007; Karypis and Kumar 1997] optimization. Some other works consider the temporal aspect when deciding partition boundaries. Shah et al. [2018] proposed speculative balancing for fluid simulation. It computes partition-to-worker assignments by performing a low-resolution simulation substitution and predicting the high-resolution workload distribution in the upcoming steps. Their partitioning overhead is

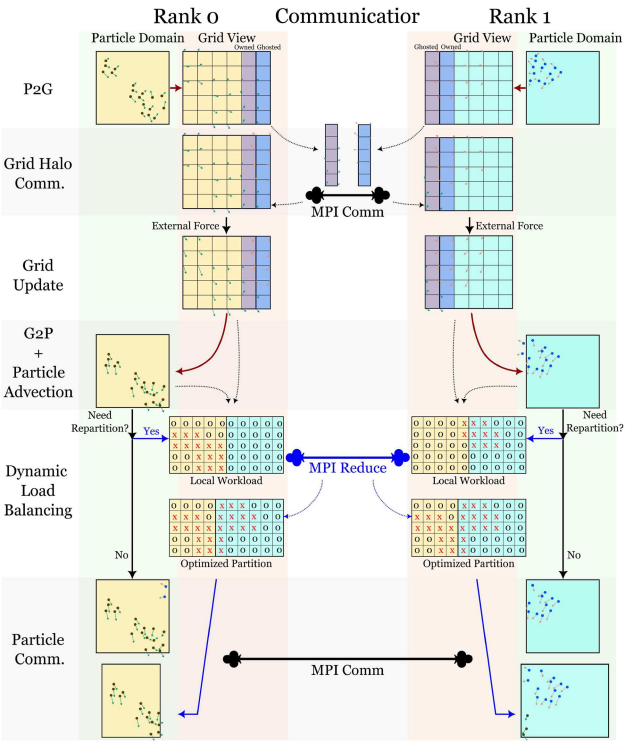

Fig. 5. Our distributed MPM simulation pipeline, using two ranks as an illustrative example. In the figure, Comm. is short for Communication.

polynomial in the number of ranks. Additionally, Qu et al. [2020] proposed a birdshot scheduling method for partitioning. It splits the simulation domain into many micro-partitions and assigns them to nodes randomly. Based on cloud computing nodes' high latency, high throughput, and full bisection bandwidth, birdshot scheduling was shown to outperform static partitioning in many fluid simulation schemes including SPH, Eulerian, and hybrid methods.

2.4 Fast MPM in Computer Graphics

MPM was introduced to graphics by Stomakhin et al. [2013] for simulating snow dynamics. Scaling MPM to higher resolution is promising since a regular Cartesian grid is used to discretize fields [Jiang et al. 2016]. Many research efforts investigated techniques to acclerate MPM. For example, Klár et al. [2017] constructed production-ready GPU MPM solvers in the Dreamworks animation pipeline with adaptive particle advection. Gao et al. [2018] studied design choices for explicit and implicit MPM parallelism utilizing GPU. Based on that, Wang et al. [2020] harnessed the power of multiple GPUs and achieved one-hundred-million-particle simulations on an eight-GPU workstation. Recently, Fei et al. [2021] summarized various principles for accelerating single- and multi-GPU MPM implementations. They achieved real-time performance for a one-million-particle simulation on four NVIDIA GPUs with NVLinks.

Taking a different path toward performance optimization, Hu et al. [2019] proposed the Taichi programming language as a high-level interface to process spatially sparse multi-level data structures. By decoupling data structures from computations, users can perform experiments using different data structures without changing much code. Hu et al. [2021] further improved this compiler by introducing low-precision numerical data types for reduced memory occupation and bandwidth consumption. It enabled faster and higher-scale simulations by sacrificing numerical accuracy.

3 BACKGROUND

3.1 Material Point Method (MPM)

MPM is a hybrid simulation method that uses particle and grid representations to discretize the simulation domain. Typically, physical attributes including mass (m_p), velocity (v_p), deformation gradient (F_p), and affine velocities (C_p) are stored on particles; grid nodes that stores mass (m_i) and momentum ($m_i v_i$), transferred from particles, are treated as auxiliary scratchpad variables to perform spatial derivative computations and boundary condition enforcement.

To demonstrate our programming model without loss of generality, we implement the most basic first-order MPM time integration scheme with the following essential steps for incremental dynamics.

(1) **Particles-to-Grid (P2G)**. Compute grid mass and momentum from particles: $\{m_p, m_p \boldsymbol{v}_p^n\} \rightarrow \{m_i, m_i \boldsymbol{v}_i^n\}$. In addition, transfer force contributions to grid nodes from elastic stresses of the nearby particles and project particle deformation gradients for plasticity (if any).
(2) **Grid Update**. Update grid velocities with either explicit or implicit time integration: $\boldsymbol{v}_i^n \rightarrow \boldsymbol{v}_i^{n+1}$, taking boundary conditions and collision objects into account.
(3) **Grid-to-Particles and Particle Advection (G2P)**. Transfer velocities from grid nodes to particles, evolve particle strains, and then update particle positions with their new velocities: $\{\boldsymbol{v}_i^{n+1}\} \rightarrow \{\boldsymbol{v}_p^{n+1}, \boldsymbol{F}_p^{n+1}\}$, $\{\boldsymbol{p}_p^n, \boldsymbol{v}_p^{n+1}\} \rightarrow \{\boldsymbol{p}_p^{n+1}\}$.

These three steps are the major computing components in MPM. We show how these computations are performed on each MPI rank in our distributed system in Figure 5.

3.2 Performance Portable Parallel Programming with Kokkos

As introduced in Section 1.2, we employ the Kokkos library [Edwards et al. 2014; Trott et al. 2022] as the device-portable *programming model layer* that supports multidimensional array allocation and access and parallel execution patterns. Using Kokkos, our simulation pipeline can switch among different backend programming models, including OpenMP and CUDA, using C++ template arguments. For example, we can make the following definitions and pass them into both particle and grid data structures and related parallel kernels to invoke an NVIDIA GPU for data management and computation. The comments show how we can quickly switch to CPU with OpenMP.

```
1 using EXECSPACE = Kokkos::Cuda; // Kokkos::OpenMP
2 using MEMSPACE = Kokkos::CudaSpace; // Kokkos::HostSpace
3 using DEVICE = Kokkos::Device<EXECSPACE, MEMSPACE>;
```

The particle data structure, our new sparse grid, and all other supporting arrays are implemented based on Kokkos::View, which defines a multidimensional array based on user-specified memory space. We can set the array size at compile time or run time. One example of defining a 2-dimensional array with Kokkos::View is listed in the first line of the code patch below. It defines an NUM×3 array, with the first dimension size (NUM) specified during run time and the second during compile time.

To dispatch parallel computations, one can use various Kokkos parallel patterns with a specified execution policy to perform defined kernels on different architectures, as shown below. In line 2, we get particle position data slices (detailed particle definitions are listed in Section 3.3), and then dispatch a Kokkos::parallel_for pattern with a range policy to assign particle positions to the pre-defined Kokkos::View. By changing the content of KOKKOS_LAMBDA, one can easily modify the behavior of the computing kernel. Finally, in line 12, we create a host copy of the View data so that the CPU-side (host-side) code can also access or further output the data to files for visualization.

```
1  Kokkos::View<T*[3], MEMSPACE> pos( "positions", NUM );
2  auto x_p = Cabana::slice<P::pos>( particles );
3  /* dispatch a parallel for to assign data from x_p to pos */
4  Kokkos::parallel_for(
5      Kokkos::RangePolicy<EXECSPACE>( 0, particle_num ),
6      KOKKOS_LAMBDA( const int idx ) {     // compute kernel
7          pos( idx, 0 ) = x_p( idx, 0 );   // element access
8          pos( idx, 1 ) = x_p( idx, 1 );
9          pos( idx, 2 ) = x_p( idx, 2 );
10     } );
11 Kokkos::fence();            // fence execution space
12 auto host_view =
13     Kokkos::create_mirror_view( Kokkos::HostSpace(), pos );
```

In addition to Kokkos::parallel_for, other parallel execution patters are supported in Kokkos, including parallel_reduce and parallel_scan. We refer to Edwards et al. [2014] and Trott et al. [2022] for further details.

3.3 Distributed Particles with Cabana

With the device-portable programming model, we are able to build the *particle/grid data layer*. As mentioned in Section 3, we utilize Cabana library for particle memory management and communication.

Built upon the Kokkos::View, the Cabana::AoSoA enables ***Array-of-Structure-of-Array*** (***AoSoA***) layout [Wang et al. 2020] to manage particle storage with user-specified properties. The *AoSoA* structure exploits the advantages of both *Structure-of-Array* and *Array-of-Structure* to conserve both coalesced threads calculations and performant random memory access patterns when parallelizing MPM. The following code sample shows how to declare particle storage with MPM-essential properties such as mass, position, velocity, deformation gradient, APIC transformation matrix, and plastic volumetric strain (lines 1–4). Additionally, lines 5–8 illustrate how to use Cabana::slice to access individual particle properties. The readers can find more details in Mniszewski et al. [2021].

```
1  using particle_members =
2      Cabana::MemberTypes<T, T[3], T[3], T[3][3], T[3][3], T>;
3  using particle_list = Cabana::AoSoA<particle_members,
       MEMSPACE>;
4  particle_list particles;
5  // access single particle properties with Cabana::slice
6  auto position = Cabana::slice<1>( particles );
7  auto velocity = Cabana::slice<2>( particles );
8  auto affine = Cabana::slice<4>( particles );
```

3.4 MPI Communication

Handling particle and grid data communication is another crucial ingredient of the *particle/grid data layer*. We use MPI [Snir et al. 1998], a message passing interface widely used for multi-node applications, to perform data communication among distinct ranks. In the rest of this article, we use MPI rank, rank, and worker as interchangeable terms of the logically independent computing unit that handles non-overlapped work.

In practice, we divide the whole simulation domain spatially and distribute the corresponding workload (particles and grids) to ranks with a user-specified MPI communicator topology. MPI ranks can be mapped to a single or multiple computing devices according to the hardware setup and the options provided when running the simulation executable with mpirun/mpiexec command. Generally, one individual process will handle one rank during execution. In our system, we use non-blocking MPI_Isend/MPI_Irecv pairs for particle and grid data exchanges among workers. Also, MPI_ALLReduce with operators like MPI_SUM or MPI_MIN are employed for inter-rank grids/particles reductions. Moreover, MPI_Barrier is called for synchronization to ensure data consistency and computation correctness.

4 DISTRIBUTED SPARSE GRID

In MPM simulations, the valid domain is generally sparsely occupied by material particles. As a result, it may cause an unnecessary waste of computing time and memory occupation in large-scale simulations if a dense grid is used. Therefore, we develop a distributed sparse grid data structure to represent the sparsely populated uniform grids in the *particle/grid data layer* to more effectively leverage the computing resources on multiple MPI ranks. Our sparse-grid approach shares kernel-level interfaces with the dense grid data structures implemented in Cabana, making it effortless for Cabana users to switch in their simulation implementations.

We distribute the simulation work to multiple MPI ranks by dividing the entire domain into rectangular partitions. Each MPI rank needs to have panoramic information to guide its local computations. Some essential global knowledge includes the size and position of the entire simulation domain and the rectangle range each MPI rank handles. To clarify the descriptions, we propose the following concepts to represent the logical simulation domain and uniform grid. Each concept is implemented as a separate C++ class in practice.

- **Global Mesh:** The actual position and size of the entire simulation domain.
- **Global Grid:** The entire logical uniform grid, indexing from 0 to the grid resolution in each dimension. The global grid also contains the domain partition information, indicating the grid range that the current MPI rank is in charge of.
- **Local Mesh:** The position and rectangle sub-domain size of the current MPI rank.
- **Local Grid:** The valid *owned* and *shared* grid indexing space of the current worker. The *owned space* represents all the grids

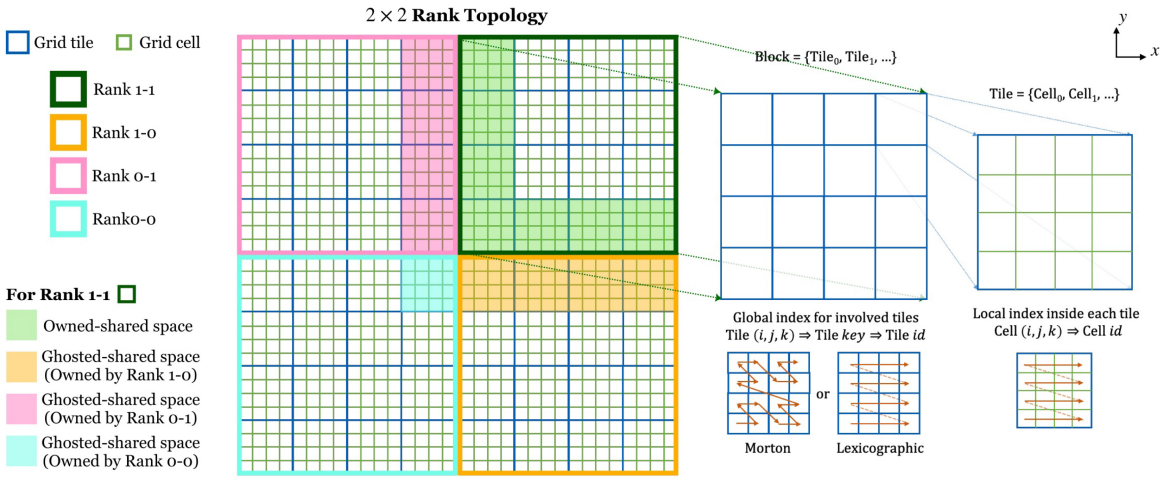

Fig. 6. **Hierarchical sparse grid representation**. We use a 2D MPI topology as an example. The entire simulation domain is divided into four *blocks*, each handled by a unique MPI rank. The *blocks* are further divided into *tiles*, which contains $N \times N$ grid *cells* (in this example, $N = 4$). Local cell indexing inside each *tile* is lexicographical. The tile ijks are mapped to 1D keys through a user-specified manner (either lexicographical or using a Morton curve). In addition, the halo regions, *i.e.*, the *shared spaces* of different MPI ranks, are classified as *owned-shared space* and *ghosted-shared spaces* as illustrated in shaded colors.

exclusively accessed by the rank, while the *shared space* indicates the halo range with which multiple MPI ranks may interact. As illustrated in Figure 6, we have two *shared space* types: (1) *owned-shared space* to represent all the grids that are owned and managed by the current MPI rank but may interact with particles residing on the neighbor ranks, and (2) *ghosted-shared space* to denote the grid range that owned by some other MPI ranks, but the current worker may read from or write to.

To further improve the flexibility of grid data management, we propose a hierarchical *block-tile-cell* representation of the simulation grid domain. *Block* is defined as the reference to the entire *local grid* domain on a single MPI rank. It is further divided into *tiles*, as shown in Figure 6, where each *tile* contains a user-defined number of *cells* ($4 \times 4 \times 4$ in our examples). This hierarchical design allows users to customize the grid data allocation and access with coalesced data access patterns that could potentially benefit parallel particle-grid interpolations. It can also fit the special design needs in user-customized simulation pipelines such as Gao et al. [2018]; Wang et al. [2020].

Before performing the grid array allocation, we define a sparse *grid layout* to specify the following information:

(1) the entity type on the sparse grid (*i.e.*, whether to store the value on grid nodes, cell centers, faces, or edges);
(2) the valid grid *tiles* in the current simulation step; and
(3) the halo status in the current simulation step.

The first piece of information is consistent throughout the entire simulation process. In practice, we define multiple overloading functions in the *local grid* concept to deal with the minor indexing and grid ownership disparity caused by different entity types. By contrast, the second and third status varies along with the simulation and, thus, require recalculation in every time step. In the following subsections, we explain how the sparsity is registered (Section 4.1) and how the halo communications are achieved (Section 4.2).

4.1 Sparse Map

In MPM simulations, the valid/activated grids, *i.e.*, the grids that will be allocated and accessed in the upcoming step, are the grids that will interact with particles. The grid range each particle will activate is determined by the particle position and the Eulerian interpolating functions. We adopt the quadratic kernel for particle/grid data transfer in all examples. Thus, we can ensure that each particle will activate only 27 grid *cells* nearby. Since we use grid *tile* as the minimum unit of actual allocation, each corresponding *tile* of these *cells* is mapped to an array index inside the grid memory by spatial hashing before MPM time integration in each step. We first map the global 3D *tile* index to a hashing key using either the lexicographical order or a space-filling Morton curve (Figure 6) [Gao et al. 2018; Setaluri et al. 2014; Wang et al. 2020] according to user's choice. This process ensures that every logically independent grid *tile* has a unique identifier on whichever worker. Then, the *tile* key is registered in a device portable hash table (Kokkos::UnorderedMap) in a specified execution manner. This way, the 3D indices of all valid *tiles* will be mapped to a linear memory span indexing from 0 to the total valid-*tile* number.

4.2 Sparse Halo

To decide whether two adjacent MPI ranks need to exchange grid data and how much to communicate, we need to consider the following factors:

— entity type stored on the grids,
— particle-grid interpolation kernel size,
— whether the grid halo region contains valid *tiles*, and
— halo size.

Concretely, the first two factors correspond to how the MPI neighbor topology is defined by the entity type and the kernel size to

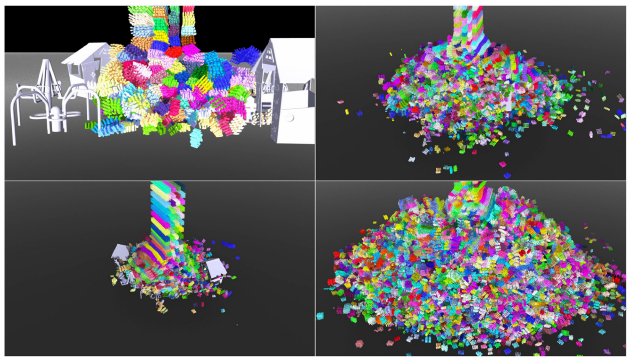

Fig. 7. **Elastic Playground.** Numerous letters, symbols, and numbers are poured onto the toy playground. At most, 394M particles are involved, and the average simulation time is 229.56 seconds per frame.

transfer particle-grid data. Specifically, in 3D with kernel size 1, the workers would share data with all 26 neighbor ranks if data is stored on grid nodes or cell centers, while only six neighbors require communication for edge and face cases. Our MPM system stores all the attributes on grid nodes with a quadratic kernel, and thus each worker needs to communicate data with all topologically adjacent ranks.

The next two factors correspond to the following. Considering the sparsity of the grid data, halo communication happens only when there are commonly registered *tiles* in the *ghosted-shared spaces* and *owned-shared space* of two neighboring ranks. And the size of the *owned-* and *ghosted- shared space* is decided by the halo size. Under this circumstance, we introduce two types of halo communications:

— *Halo Scatter.* Scatter the data in the *ghosted-shared space* of the current MPI rank to their owner rank and perform the specified grid reduction (such as summation or computing the minimum/maximum value). Note that the reduction happens on the *owned-shared space* of the owner worker.
— *Halo Gather.* Gather grid attributes in the *ghosted-shared space* of the current worker from the *owned-shared space* of the neighboring owners.

The *halo scatter* happens after the *P2G* transfer in MPM time integration. The grid owner ranks collect and reduce all valid grid data during this process. Afterwards, all owner ranks will contain complete grid information transferred from simulation particles, including in *owned space* and *owned-shared space*s. Then, *halo gather* is performed before *grid update* to ensure all MPI ranks hold the entire and correct grid data in *shared space*s.

To reduce the MPI communication overhead, we first count the valid *tiles* in the *ghosted-* and *owned-* shared space before *halo gather/scatter*, and broadcast the counting results to the neighbor ranks. For *halo scatter*, workers will send halo data to a specific neighbor only if both the counting in its *ghosted-shared space* and the counting in the neighbor's *owned-shared space* are non-zero. Additionally, a worker will wait to receive data from a neighbor only when the *owned-shared space* and the corresponding neighbor's *ghosted-shared space* are non-empty. Similar verification is also performed before actual data transfer in *halo gather* operation, with the role of *ghosted-* and *owned-* shared space switched.

4.3 Sparse Array Allocation

Based on the information provided by the *grid layout* (specifies entity type, grid activation, and sparse halo), the sparse *grid array* is created and allocated. The grid is managed in an *AoSoA* manner, with each *tile* serving as a basic *Structure-of-Array* unit, i.e., the *cell* properties inside each *tile* are organized in an *SoA* manner while the *tile structures* are listed in an outer array. By controlling the *tile* size, user can switch grid data to either *SoA* (when *tile* size equals to the *block* size) or *AoS* (when *tile* size equals to $1 \times 1 \times 1$ *cell*). As proposed in Wang et al. [2020], this design helps improve data vectorization inside a contiguous array of member variables and overall device cache efficiency with small grid tiles.

The following code example shows how to define and allocate the sparse grid in the proposed programming model. In line 1, we specify the primary value type of grid attributes. Moreover, in lines 2–3, we define the attributes stored on grids by listing all member types (mass (1D) and grid momentum/velocity (3D)). Users can easily adjust data channels by modifying the template definitions. Then, in lines 4–5, we create a sparse map (Section 4.1) to record valid grid *tiles* in each simulation step. Here, the MEMSPACE indicates whether the hashing data is on CPU or GPU and further decides whether the hash insertions or queries are performed or paralleled within the host or device kernels.

```
1  using T = float;     // or other types like double
2  using node_members = // mass and momentum in MPM simulation
3    Cabana::MemberTypes<T, T[3]>;
4  auto sparse_map =    // hash table, Sec 4.1
5    Cajita::createSparseMap<MEMSPACE>(global_mesh, reserve_size);
6  /* create grid array layout edwards2014kokkos, contains sparse
       halo (Sec 4.2); the entity type Cajita::Node() indicates
       values are stored on grid nodes */
7  auto layout =
8    Cajita::createSparseArrayLayout<node_members>(local_grid,
       sparse_map, Cajita::Node());
9  auto nodes =         // allocated grid AoSoA
10   Cajita::createSparseArray<DEVICE>("nodes", layout);
11 nodes.reserve(pre_allocate_cell_num);   // optional
```

Later in lines 7–8, we need to specify the *grid layout* from the *local grid*, *sparse map*, and the entity type to support the actual array allocation. In detail, entity type guides the halo communication and *array* allocation (Section 4); while *local grid* computes the *owned-/ghosted- tile* ranges. Note the *tile* ranges in *shared space*s require update once the simulation domain is partitioned (Section 5) to ensure the communication correctness. Finally, an *AoSoA* array is created with the pre-prepared information in lines 9–10. Automatic reallocation will be triggered during simulation if the valid grid *array* size exceeds the allocated capacity. We recommend explicitly reserving spaces for grid data by providing an estimation of the maximum valid cell number (line 11) to reduce performance drop caused by unnecessary reallocation.

5 DISTRIBUTING AND LOAD BALANCING

For performance-portable large-scale simulations, evenly distributing the workload to multiple ranks is essential. Considering the significance of load balancing, simulation communities have formulated the partitioning as a domain optimization problem [Surmin et al. 2015]. However, most existing works focus on the theory and formulation. In this section, we propose and demonstrate a detailed dynamic load balancing algorithm as an essential component of most distributed computing systems. The corresponding

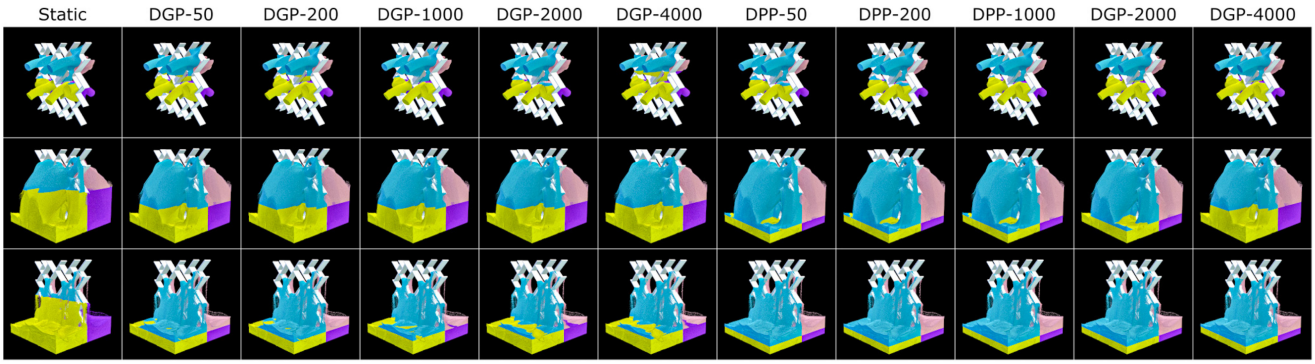

Fig. 8. **Sand injection** with different partition algorithms (133.2M particles in total, grid resolution 256×256×256). The first column shows static partitioning result, and *DGP-N* refers to *Dynamic Grid Partitioning* per N simulation steps. Similarly, *DPP* refers to *Dynamic Particle Partitioning*. Row 1 to 3 shows the results for frames 25, 99, and 145, respectively. Different color refers to particles (simulation sub-domain) handled by different MPI ranks.

ALGORITHM 1: Dynamic Load Balancing

Input: Sparse map *map* ▷ for dynamic grid partitioning
Input: Particle positions pos_p ▷ for dynamic particle partitioning
Input: MPI communicator *comm*
Output: Optimized partition $P = \{I, J, K\}$
Output: Optimization iteration times performed n

```
computeLocalWorkload(map or pos_p)        ▷ Section 5.1.1
computeGlobalWorkload(comm)               ▷ Section 5.1.2
computePrefixSum                          ▷ Section 5.1.3
n ← 0
while n < n_max do                        ▷ n_max: max iteration time
    dim_sequence ← random permutation of {0, 1, 2}
    for all d ∈ dim_sequence do
        is_changed ← false
        is_dim_changed ← optimization1D(d)
        is_changed ← is_changed || is_dim_changed
    end for
    n ← n + 1
    if NOT is_changed then
        return n
    end if
end while
```

implementation is integrated into our *particle/grid data layer*, *i.e.*, into the Cabana library. It will be fully open-sourced with detailed documentation and unit tests. In the following article, we first introduce two definitions of simulation workload in Section 5.1, and then explain our 3D partition optimization in Section 5.2. The complete dynamic load balance optimization algorithm is summarized in Algorithm 1.

5.1 Workload Computation

As introduced in Section 4, the entire work is distributed by partitioning the simulation domain into non-overlapped rectangular sub-regions according to the MPI topology, with every independent MPI rank handling each sub-region. To perform the partition optimization, we need to evaluate the workload on each MPI worker, *i.e.*, inside each rectangular sub-region, which changes dynamically throughout the simulation. In addition, the optimization process requires frequent workload analysis of the partitioned attempts. Thus, we need a representation that supports efficient workload computation within any rectangle regions.

We construct a 3D matrix to realize this goal, with each element referring to the workload value inside the corresponding area. The granularity of the workload matrix influences the accuracy and performance of the load balancing optimization. A matrix with elements representing smaller-sized regions helps the optimizer to make a more accurate and flexible choice but may increase the computation and communication overhead.

5.1.1 Local Workload Computation. First, we count the workload handled locally on each MPI rank. In hybrid simulation methods, particles and grids are two crucial representations. Thus, both can measure the work amount, leading to two types of workload computing methods.

Particle-based workload computation. In this case, each particle is treated as a work unit. We make a parallel loop over particle positions and perform atomic addition to corresponding elements in the workload matrix. This method finally leads to a partition where all MPI ranks contain a similar number of particles. Generally, hybrid methods use dramatically more particles than valid grids (typically, each grid *cell* contains at least eight particles in 3D, and sometimes more to increase details and reduce numerical fractures). Thus, computations involving particles and atomic additions consume more computing and time resources for workload statistics. Nevertheless, balanced particle distribution can potentially benefit the timing of particle-grid data transfer if particles per cell are similar all over the domain because particle number decides the number of parallel kernels and majority memory accesses.

Grid-based workload computation. In order to improve the load balancing time efficiency, we also support the workload computation based on valid grid *tiles*. The compute kernel loops over the hash table in the *sparse map* and sets the workload matrix element to 1 if the *tile* is valid, i.e., no atomic additions are required, and fewer matrix elements are involved compared to particle-based computation.

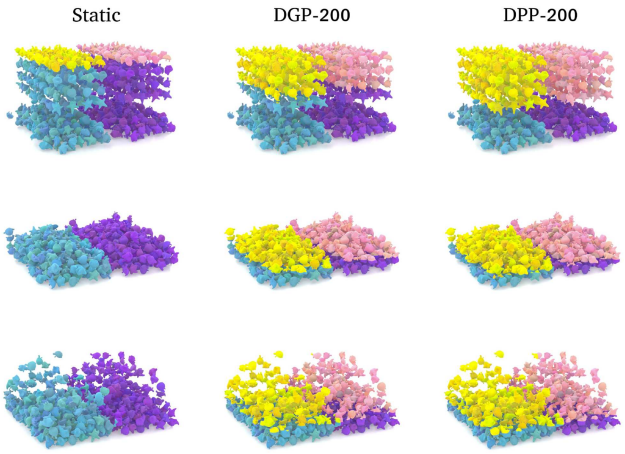

Fig. 9. **Elastic toys** with different partition algorithms distributed on 4 ($2 \times 2 \times 1$) MPI ranks (22.4M particles in total, grid resolution $256 \times 256 \times 256$). Row 1 to 3 shows the results for frames 6, 12, and 22, respectively. Particles are colored yellow, blue, purple, and pink to indicate the MPI ranks they belong to. Minor hue differences are applied to separate toys.

Discussion on the choices. Generally speaking, both methods estimate the workload distribution from different perspectives in a given domain. When particles are relatively evenly distributed in grids, *e.g.*, in elastic simulations, these two representations will generate similar load balancing results. In this situation, grid-based workload outperforms as it uses fewer computing resources. However, in simulations for granular media and fluids, the particles can splash out dramatically or gather locally. In this case, particle-based method standouts because valid grid *tiles* are likely to contain significantly different numbers of particles, leading to distinct *P2G* and *G2P* time and memory requirement on different ranks. This may influence the simulation performance, as demonstrated in Section 7, or cause run-time particle memory allocation errors after *particle communication* for large-scale scenes.

In the rest of this article, we refer to the load balancing algorithm as ***dynamic grid partitioning*** **(DGP)** if the workload is computed from the valid grid *tiles*, and as ***dynamic particle partitioning*** **(DPP)** for the particle-based case. See Section 7 for detailed comparison results.

5.1.2 Global Workload Computation. All MPI ranks need to know the workload distribution in the whole simulation domain to perform global optimization. Thus, we need to gather all the computed local workload matrices to form a global matrix. We achieve this calculation by performing MPI reduction among all ranks with the MPI_Allreduce interface. Note that CUDA-aware MPI is required if the simulation uses CUDA memory and GPU execution space.

5.1.3 Global Workload Prefix Summation. We must scan all dimensions to perform load balancing optimization and analyze if the current partition is optimal. This process requires frequent workload counting inside any arbitrary rectangle sub-regions. Inspired by Surmin et al. [2015], we compute the 3D prefix summation of the global workload matrix, pursuing a constant-time workload estimation. Specifically, we adopt Kokkos::parallel_scan interface as an efficient solution for dispatching parallel inclusive/exclusive scans with a user-defined functor and a parallel-execution policy. We scan the 3D workload matrix in three dimensions separately to compute the 3D prefix summation matrix, i.e., the first scan is in the x direction, and then the second and third scans are based on the intermediate matrices in y and z direction individually. The concrete algorithm is listed in the supplemental document.

5.2 Partition Optimization

In this section, we first summarize the formulation of the 3D partition optimization process and then introduce the detailed algorithm we used for dynamic load balancing implementation. We use I, J, K to represent the partition in dimension x, y, and z, with $I = (i_0, i_1, \ldots, i_{N_x}), J = (j_0, j_1, \ldots, j_{N_y})$, and $K = (k_0, k_1, \ldots, k_{N_z})$ indicating the dividing boundary sets. Here, N_x, N_y, and N_z are the total number of MPI ranks in corresponding dimensions, and i_*, j_*, k_* refers to the *tile* indices. Specifically, $i_0 = j_0 = k_0 = 0$ and $i_{N_x}, j_{N_y}, k_{N_z}$ equals to the total number of *tiles* in x, y, and z dimension, respectively. Note that the workload matrix granularity will influence the unit of the pre-mentioned indices in the proposed implementation. In practice, to make implementations easily understandable and consistent, we use grid *tile* as the atomic unit of (1) workload matrix, (2) partition boundary index, and (3) grid data communication.

For any given rank (α, β, γ), the local grid domain it in charges is given by grid *tiles* $\{(i, j, k) | i_\alpha \le i < i_{\alpha+1}, j_\beta \le j < j_{\beta+1}, k_\gamma \le k < k_{\gamma+1}\}$. Suppose the starting *tile* of the current rank are optimized and fixed; the proposed load balancing algorithm will find the optimal ending *tile* indices by solving the following optimization.

$$\min_{i_{\alpha+1}, j_{\beta+1}, k_{\gamma+1}} : \sum_{i=i_\alpha}^{i_{\alpha+1}} \sum_{j=j_\beta}^{j_{\beta+1}} \sum_{k=k_\gamma}^{k_{\gamma+1}} |W_{i,j,k} - \overline{W}|$$

Here, $W_{i,j,k}$ is the workload in *tile* (i, j, k) and $\overline{W}$ refers to the average rank workload computed by $\frac{\sum_{i=0}^{N_x} \sum_{j=0}^{N_y} \sum_{k=0}^{N_z} W_{i,j,k}}{N_x \times N_y \times N_z}$.

As discussed in Surmin et al. [2015], this optimization is an NP-complete problem. With previous partitions $(i_0, \ldots, i_\alpha)$, $(j_0, \ldots, j_\beta)$, and $(k_0, \ldots, k_\gamma)$ fixed, there are three degree-of-freedoms to decide, *i.e.*, the optimal partition $i_{\alpha+1}, j_{\beta+1}$, and $k_{\gamma+1}$, for the current rank. In addition, the results will influence the computation for later ranks with larger rank indexing values. To solve this problem with three unknowns, we iteratively alternate among each variable and perform 1D optimizations. The iteration will stop when the partitioning results are unchanged or the maximum iteration number is reached, as shown in Algorithm 1. In the supplemental document, we present a validation example to show that this iterative algorithm can generate the optimal solution with several iterations when the ground truth is unique.

5.2.1 1D Load Balancing Optimization. Inside each 1D optimization, we randomly choose one dimension of interest that is never covered in the current iteration. This randomness reduces the possibility for the algorithm to be trapped into local optimal and potentially reduces the iteration times. Then, the partition in

the non-chosen two dimensions is fixed. All partition boundaries in the dimension-of-interest will be reanalyzed individually for a more even workload division. The detailed algorithm is summarized in the supplemental document.

In practice, it is possible to have a range of consecutive *tiles* where there are no valid particles. In theory, any *tile* indices in this range can be treated as optimal partition positions. However, because dynamic load balancing is not performed in every simulation time step, if we choose the pre-mentioned *tile* range boundaries as the partition position, the particles may move over the range boundaries before the next round of partitioning. This choice will cause extra particle communications in the upcoming steps, especially for solid simulations, where many particles tend to gather together, and the particle communication overhead would be considerable. Therefore, we always set the partition point as the middle point of the equivalent *tile* range where there are no particles. This simple operation reduces the potential particle communications among MPI ranks and improves the overall performance.

6 DISTRIBUTED MPM IMPLEMENTATION

6.1 Time Integration

As mentioned in Section 3, we implement the first-order MPM time integration scheme including three basic *computation kernels*, *i.e.*, **P2G**, **Grid Update**, and **G2P**. For distributed systems, another two *communication kernels*, **Grid Halo Communication** and **Particle Communication** are required to guarantee the correctness. In detail, **Grid Halo Communication** is performed before *Grid Update* to ensure the completeness of the grid data on each MPI worker. It consists of the *halo scatter* and *gather* operations introduced in Section 4.2. Then, after updating particle positions in **G2P**, we end up time integration step with **Particle Communication** to distribute particles to the ranks in charge of the corresponding grid sub-domain. Additionally, **Dynamic Partition Optimization** is performed right before *Particle Communication* at certain time steps to ensure a relatively balanced load distribution. The entire pipeline is illustrated in Figure 5.

6.2 PIC Algorithms and Application Implementation

To make a more complete distributed MPM simulation system, we add the *PIC Algorithm layer* to further support the *application layer* in building up versatile large-scale scenes. In addition to the core time integration routines, we add components like device-portable sparse collision object, particle sourcing, analytic/VDB-based shape, and multi-material constitutive modeling with elasticity and plasticity, forming the MultiSim library (Figure 1). We support four types of constitutive models, including fixed-corotated model [Stomakhin et al. 2013] for elasticity, Drucker-Prager elasto-plasticity [Klár et al. 2016] for sand simulations, **Non-Associated Cam-Clay** (**NACC**) [Li et al. 2022; Wolper et al. 2019] for snow/mud-like behaviors, and furthermore, weakly compressible fluids [Tampubolon et al. 2017] for liquids. Users can easily specify the component or even extend the current *PIC Algorithm layer* for more applications. In the supplemental document, we provide a concrete example showing how to use the components in *application level*. More demos can be found in our open-source code.

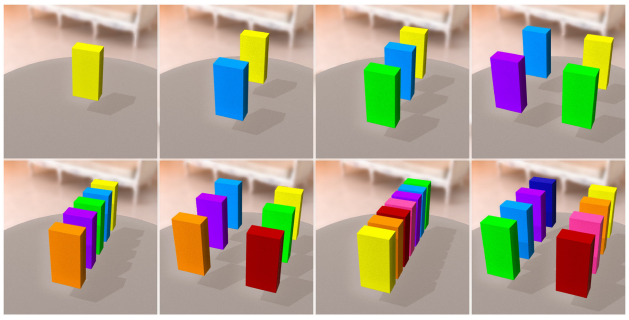

Fig. 10. **Weak Scalability** scene setup with 1–8 workstations. Different colors refer to different MPI ranks in each sub-figure.

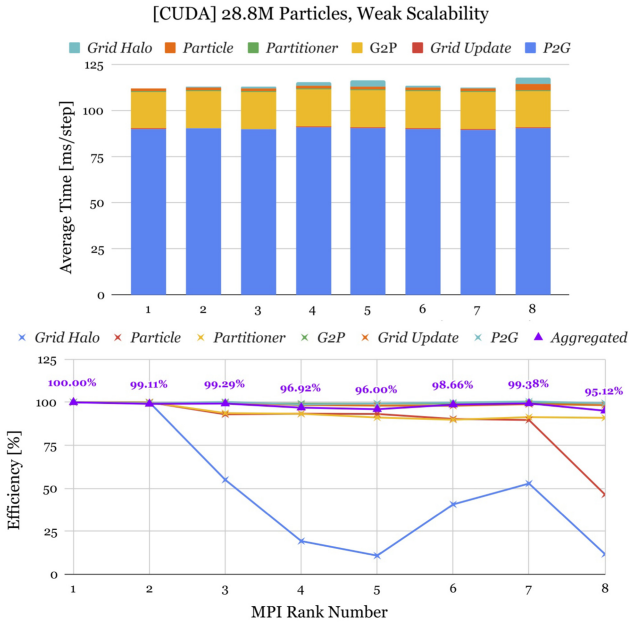

Fig. 11. **Weak Scalability** on local workstations with GPU(CUDA). Here, *Particle* is the short for *Particle Communication* kernel; and similarly, *Grid Halo* for *Grid Halo Communication* and *Partitioner* for *Dynamic Partition Optimization*. The listed numbers in the lower figure are the efficiency values for aggregated timing (summation of all six kernels).

7 RESULTS AND EVALUATIONS

This section evaluates the proposed distributed MPM framework with scaling tests, load balancing comparisons, MPI Cartesian topology comparisons, and large-scale demonstrations. We use at most eight workstations (each as one MPI rank) in our experiments. The workstation has one Intel Core i9-10920X (12 core, 24 threads, base clock 3.50Hz) and one NVIDIA GeForce RTX 3090 GPU. We adopt 10-Gigabit bandwidth Ethernet to support inter-rank communications, with OpenSSH [developers 2021] and CUDA-aware OpenMPI 4.1.2 [Members 2021]. All the evaluations and demonstrations are conducted under this setup unless stated otherwise.

7.1 Multi-MPI Scalability

Scalability with increased computing resources is a widely adopted test to evaluate the effectiveness and robustness of a distributed

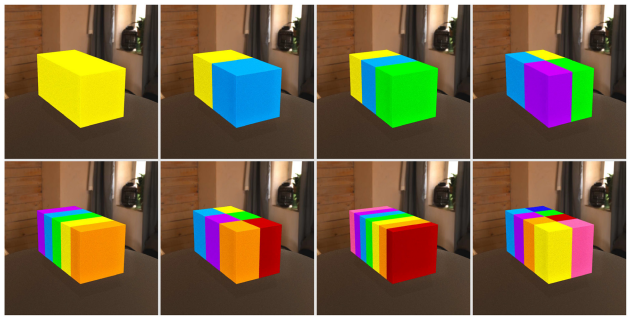

Fig. 12. **Strong Scalability** scene setup with 1–8 workstations. Different colors refer to the particles handled by different MPI ranks in each sub-figure.

algorithm. Ideally, performance should scale up with the number of involved MPI ranks. However, a perfect scaling is not practical. Specifically, Amdahl's law and Gustafson's law demonstrate the limitation of parallel computing; furthermore, the communication bandwidth also constrains the upper bound of multi-rank acceleration. To analyze the performance of the proposed distributed MPM system, we present the scaling results on local workstations with CUDA as a latent programming model. Additionally, experiments distributed with OpenMP are summarized in the supplemental document.

7.1.1 Weak Scaling. Inspired by Gao et al. [2018], we set up the experiment by placing an elastic cuboid at the center of each rank's *local mesh* and let it fall with gravity, as illustrated in Figure 10. All cuboids are of the same size with 28.8M particles. The MPI rank topology is $n\times1\times1$ for rank number $n = 1, 2, 3, 5, 7$, and $n/2\times1\times2$ for $n = 4, 6, 8$. We summarize the experiment timing and efficiency of each computing/communication kernel in Figure 11. For communication kernels (*Particle Communication*, *Dynamic Partition Optimization*, and *Grid Halo Communication*), efficiency is computed with 2-rank timings as the 100% base, since there's little communication overhead for the 1-rank case. As demonstrated, the aggregated weak efficiency is over 95% regardless of the rank numbers. To further illustrate the scaling potential of the proposed model, we run the test with up to 120 ranks (20 nodes) with CUDA on the Summit supercomputer and show the results in the supplemental document.

7.1.2 Strong Scaling. For the strong scaling test, we assign a falling cuboid at the center of the *global mesh* as a fixed-size problem and bring different numbers of MPI ranks into the computation. The cuboid contains 159M particles for the CUDA test. The entire workload is automatically divided and assigned to ranks by applying the proposed dynamic load balancing algorithm given a user-specified MPI topology, as shown in Figure 12. We conduct the experiments with the same MPI rank topology settings as in Section 7.1.1. The timing and speedup analysis are illustrated in Figure 13. Our system can pursue an almost linear overall speedup as the MPI rank increases.

7.2 Load Balancing Studies

Dynamic load balancing generally boosts the simulation performance of a distributed system from several perspectives, as stated before. However, partition optimization and the commensurate

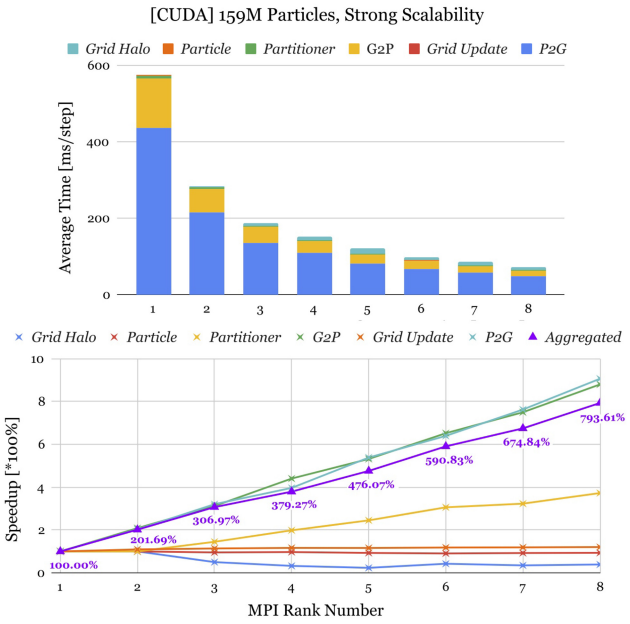

Fig. 13. **Strong Scalability** on local workstations with GPU(CUDA). All ranks handle a huge elastic box with 159M particles.

particle relocation may require significant computation and communication time. In addition, various material behaviors may lead to divergent partition results when using different workload elements. Therefore, we conduct several experiments with multiple material behaviors to evaluate the proposed load balancing algorithms in this section. The results and discussions can help users find the best choice for their simulation objectives.

7.2.1 Sand Injection. In this experiment, we focus on comparing the behavior of the *static*, *dynamic grid*, and *DPP* methods and analyzing how partitioning frequency influences the performance. As displayed in Figure 8, we design a 4-MPI-rank *sand injection* scene, where each rank injects sand from two sourcing points with random velocities pointing toward the shelf (collision object) sitting at the domain center. Throughout the simulation, sand material sometimes splashes and finally settles down, leading to dynamically varying workload distribution.

Dynamic Partitioning V.S. Static Partitioning. For a thorough analysis, we illustrate the detailed timing on all four ranks for *static*, *dynamic grid*, and *dynamic particle* partitions in Figure 14. In addition to the timing of each separate kernel, we also show waiting time, which refers to the duration when faster ranks finish computation/communication and wait for other ranks. We show the data with dynamic partitions performed every 50 steps without loss of generality. In the first row of Figure 14, static partitioning pushes more work to lower ranks (rank 0-0-0 and 1-0-0) as the sand particles fall to the ground. The upper ranks (rank 0-1-0 and 1-1-0), on the other hand, contain fewer and fewer particles and thus sit idle, wasting time waiting for the lower ranks. This issue is mitigated when dynamic partition is adopted (rows 2–3 in Figure 14).

In this test, some sand particles splash out in the upper subdomains while the others pile up at the bottom. This uneven

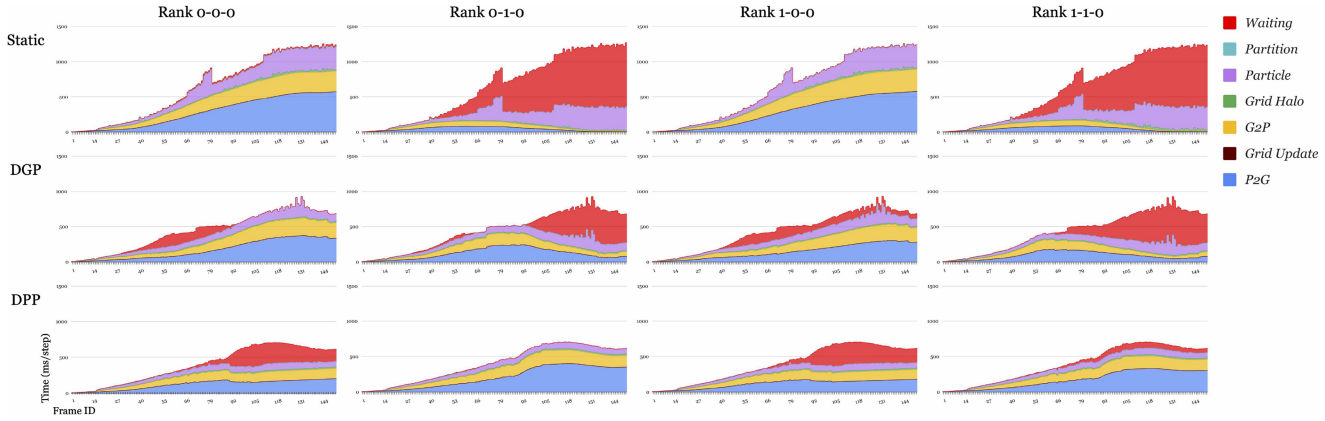

Fig. 14. Detailed timing per step (in milliseconds) of sand injection with *static partitioning*, DGP, and DPP.

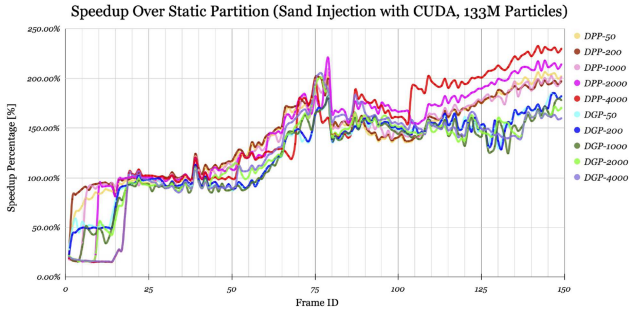

Fig. 15. Aggregated time speed up of dynamic load balancing over *static partitioning*. *DGP-N* and *DPP-N* refer to *dynamic grid partitioning* and *dynamic particle partitioning* performed every N steps, respectively.

particle-per-grid-*tile* distribution leads to different behaviors of *DGP* and *DPP*. When applying *DGP*, each rank contains a similar number of grids, but the upper ranks need to handle more particles. This fact means that more parallel work is required for upper ranks to perform *P2G* and *G2P*, and thus the lower ranks become idle, especially after frame 90. For *DPP* the roles reverse as the lower ranks need to handle more of the grid, making the upper ones wait.

Dynamic Partition Frequency. We compare the speedup of *dynamic grid/particle partitioning* with different frequencies to static partitioning in Figure 15. This specific simulation takes around 233 steps per frame, and the sand particles are continuously injected until frame 80. We choose partitioning step intervals to be 50, 200, 1000, 2000, and 4000 for testing, *i.e.*, performing dynamic load balancing per about 0.25, 1, 4, 8, and 17 frames.

As illustrated in Figure 15, all choices achieve over 1.4x speedup and can reach 2.3x in some frame ranges. In theory, the best speedup would be about 2x, as the extreme case is that the lower two ranks handle all workload and the upper two do nothing but wait. This speedup can be better in practice when considering the overhead of parallel scheduling, memory access, and communication.

Overall, *DPP* outperforms the grid-based method for the splashing materials. Moreover, each partitioning frequency has different speedup trend through frames 0–25, 25–80, and 80–150. This indicates that the particle/grid number (problem scale), material

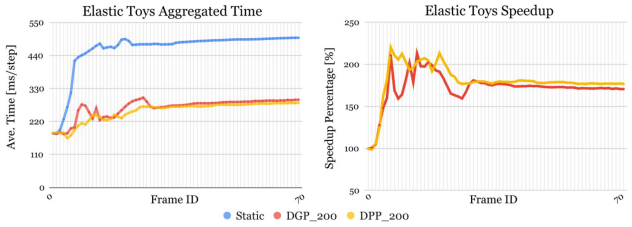

Fig. 16. **Elastic Toys.** (Left) Aggregated simulation time statistic, averaged on each step. (Right) Speedup of dynamic partitioning over the static partitioning.

Table 1. **Sand Inject Speedup**

Method	Static	DPP-50	DPP-200	DPP-1000	DPP-2000	DPP-4000
Time (h)	**6.15**	3.83	3.82	3.83	**3.62**	3.64
Speedup (%)	–	160.49%	160.87%	160.30%	***169.89%***	169.10%
Method (h)	–	DGP-50	DGP-200	DGP-1000	DGP-2000	DGP-4000
Time (h)	–	4.26	**4.25**	4.37	4.25	4.26
Speedup (%)	–	144.28%	***144.68%***	140.69%	144.49%	144.13%

We summarize the total simulation time of the 150 frames in hours and the speedup of partition methods with different partitioning frequencies. *DPP* and *DGP* achieve the best overall speedup 2000 and 200 steps, separately.

behavior (sourcing, splashing, and falling), as well as the motion (if particles are moving toward the same direction as the partition boundaries) will all influence the actual performance. Despite the partitioning frequency, our dynamic load balancing algorithm, compared to the static case, can always accelerate the simulation process as summarized in Table 1, and it can gain more speedup for large-scale cases that consume more time (after frame 80 when sourcing stops as in Figure 15).

In particular, we observe that *DPP* per 4000 steps behaves better than other cases after frame 80. There are two possible reasons. First, frequent partition changes prompt immediate particle relocation among ranks. It will also introduce extra *particle communication* work in the following steps, especially when particles and partition boundaries move in the same direction. Second, a relatively perfect particle partition leads to an undesirable grid partition for splashing sands. Nevertheless, delayed partitioning alleviates this situation by pushing more particles to lower ranks but more grid to the upper ranks, thus

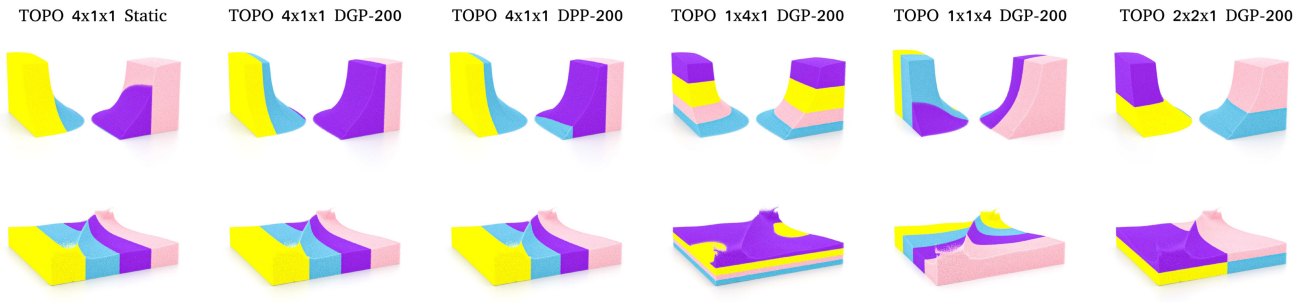

Fig. 17. **Sand dambreak** with different partition algorithms and MPI topology settings (17M particles in total, grid resolution $256 \times 256 \times 256$). TOPO in the figure refers to MPI Cartesian topology. Rows 1 and 2 show the results of frames 21 and 49, respectively. Different color refers to particles (simulation sub-domain) handled by different MPI ranks.

leading to more rank-balanced particle-grid computations. *DGP*, however, cannot benefit from this partitioning delay.

7.2.2 Elastic Toys. This experiment shows how partition methods behave when materials splash considerably less. Initially, we assign four MPI ranks the same number and type of toys and thus the same amount of particles, and we drop them as shown in Figure 9. With toys falling down, particles are communicated to lower ranks if they pass the upper-lower partition interface. Here, the overall acceleration rates (173% for *DGP* and 180% for the particle case) are similar and are close to the best theoretical speedup (200%). In a detailed timing statistics (Figure 16), we notice better acceleration with *DPP* before frame 30. It happens because the toys are of irregular shape and random orientation. Thus, some active grid *tiles* contain only a sharp toy corner with few particles. Lower toys reach the bottom while falling, but the upper ones are still placed evenly in the sky. As a result, more grids will be activated in the upper domain, leading to a partition boundary closer to the upper toy group. The toy's falling direction makes the workload less balanced in the steps prior to the next round of partitioning. One of the representative frames is shown in the first row of Figure 9.

7.2.3 Sand Dam Break. Another classic scene we test is sand dam break illustrated in Figure 17. Initially, two sand columns are located at the diagonal corner of the entire domain. Unlike previous settings, we use MPI rank topology $4 \times 1 \times 1$ to evaluate the behavior of dynamic load balancing algorithms. In this simulation, the sand flows toward the domain center and settles down onto the floor at last. There is no splashing or extreme deformations throughout this process. Thus the two dynamic partition methods achieve similar speedups as demonstrated in Figure 18. In addition, with more and more sands gathering into the middle domain, *static partitioning* gradually becomes a naturally appropriate approach (after frame 50). In this case, the dynamic load balancing methods exhibit only 10%–15% performance improvement.

7.3 Cartesian Topology of MPI Ranks

Here we observe another factor that strongly influences the performance of distributed simulators: the initial MPI Cartesian topology setting. In this section, we use the *Sand Dam break* example introduced in Section 7.2.3 for illustration. We rerun the simulation with MPI topology $1 \times 4 \times 1$, $1 \times 1 \times 4$, and $2 \times 2 \times 1$ (Figure 17) and compare the timing to the case $4 \times 1 \times 1$. To make a fair comparison, we

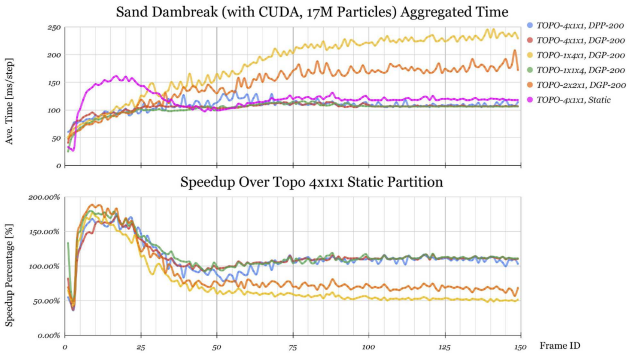

Fig. 18. **Sand Dambreak.** (Top) Aggregated simulation time statistic, averaged on each step. (Bottom) Speedup over static partitioning with MPI rank topology $4 \times 1 \times 1$.

adopt *DGP* for all the new topology settings and summarize the timings in Figure 18.

With identical computing resources, the simulation performance reduces significantly with inappropriate MPI topologies ($1 \times 4 \times 1$ and $2 \times 2 \times 1$). There are several reasons leading to this result in this specific simulation scene. First, sands move with gravity toward the $-y$ direction. If there are rank boundaries on the y dimension, particles must be relocated to other ranks when flowing down, increasing the *particle communication* overhead. Second, diverse rank topologies lead to a different number of neighbors and the size of halo areas, causing performance variance. Consequently, the takeaway is that we should always carefully consider the particle distribution and motion tendency in the scene to set the initial MPI topology for the best performance.

7.4 Large-Scale Simulations

This part demonstrates the scalability of the proposed distributed MPM scheme with a suite of large-scale simulations. The corresponding settings and average time per frame are summarized in Table 2, while the precise kernel timings are in the supplemental document.

1B-Fluid. Example in Figure 2 exemplifies a complex fluid scenes containing 1.01 billion particles falling onto chip-shaped boards. To the best of our knowledge, this is the first MPM simulation beyond the scale of 1B. Initially, there is a water layer on the ground

Table 2. **Parameters and Timings**

Example	Particle #	Rank #	Grid Resolution	Ave sec/frame	Δt_{frame}	Δx	max Δt_{step}	material parameters
(Figure 2) 1B-Fluid	1, 006, 766, 992	8	$256\times266\times256$	323.25	1/4	100/256	1.01×10^{-3}	Fluid: (500)-(3×10^6, 3)
(Figure 3) Mudflow	207, 810, 349	4	$256\times256\times256$	159.34	1/10	200/256	3.36×10^{-4}	NACC: (1500, 1.5×10^7, 0.3)-(−0.007, 0.05, 30, 30)
(Figure 4) High-resolution Sand Injection	266, 507, 608	4	$512\times512\times512$	1, 072.95	1/60	1/512	3.58×10^{-5}	Sand: (20, 1×10^4, 0.4)-(30.0, 0.0)
(Figure 7) Elastic Playground	393, 954, 516	4	$512\times512\times512$	229.56	1/4	200/512	8.42×10^{-4}	Fix-corotated: (1000, 9×10^6, 0.4)

We summarize the parameters of particle numbers, grid resolutions, MPI rank numbers, grid *cell* size Δx, and the average time per frame for various experiments described in Section 7.4. The material-related parameters are listed as well. In addition to the basic material settings (density ρ, Youngs Modulus E, and Poisson Ratio ν), parameters needed by specific materials are provided. We refer to the corresponding papers for a physical explanation of the parameters. For *fix-corotated* model, we simply show (ρ, E, ν); while for NACC, parameters are given with format (ρ, E, ν)-(α_0, β, ξ,friction angle); sand model (Drucker-Prager elastoplasticity) includes parameters (ρ, E, ν)-(friction angle, cohesion), and finally, we provide (ρ)-(k, γ) for fluids.

Table 3. **CPU/GPU SOTA Comparison**

Device	Method	Particle #	Timing (ms/step)
CPU	[Wolper et al. 2019]	10M	1034.98
	Ours	10M	2261.39
GPU	[Gao et al. 2018]	20M	39.89
	Ours	20M	73.17

Note that the particle number is rounded.

and eight liquid cubes falling on top. We use eight workstations to solve this challenging problem, with MPI topology 4×1×2 through CUDA parallelization. In order to maintain a balanced fluid distribution, we use the *DGP* (per 50 steps). As a result, every MPI rank consistently uses 22 to 23 Gigabytes of GPU memory over the span of the simulation. The fluid's turbulence is recorded in a large amount of detail, as shown in Figure 2.

Mud flow. We simulate natural mud flow (Figure 3) with NACC models. The domain is of size 200×200×200 meters, with a bumpy slope serving as a collision object. Mud particles are injected into the scene in the first 500 of the total 1000 frames and reach 207.8M at maximum. The simulation is performed with 4×1×1 MPI ranks, and *DGP* every 200 steps. Figure 3 also visualizes the mud flow damage propagation.

High-resolution Sand Injection. Figure 4 demonstrates the scalability with a high-resolution version of *Sand Injection*. We increase the spatial resolution to 512 × 512 × 512, and the scene reaches 266.5M particles at the middle point of the simulation time. Compared to Figure 8, there are more splashing and collision details in Figure 4. This demo has the same MPI rank topology as the low-res case.

Playground. Another scene involving more than 393.95M particles shows elastic jellos spreading onto the ground. We visualize some frames from different viewpoints in Figure 7. In this test, numerous elastic letters, numbers, and symbols are continuously poured into a toy playground. Four workstations (4 × 1 × 1) are adopted for computation.

7.5 Single-Machine Performance

For completeness, we also compare our distributed MPM pipeline with the state-of-the-art (SOTA) C++ MPM simulation pipeline implementations that are heavily hand-optimized for single architectures: CPU [Wolper et al. 2019] and GPU [Gao et al. 2018]. As the test example, we use a simple scene, an elastic box falling down to the ground, with grid resolution 256 × 256 × 256. Table 3 summarizes the particle number, testing device, and timing results. Indeed, our distributed system cannot outperform the separate CPU- and GPU- implementations, which have CPU-tailored SPGrid [Setaluri et al. 2014] and CUDA kernel optimizations [Gao et al. 2018], but is near 50% performance for both. Nevertheless, our target is a generalized distributed and scalable system that can be executed on most HPC platforms without code modification, while the compared implementations have adopted sophisticated hardware specific code optimizations focusing on their specialized single-machine platforms.

8 CONCLUSION AND DISCUSSION

We proposed a distributed simulation framework specialized for MPM computations. Based on the hierarchical system architecture design, we make it possible to achieve multiple advancements in each layer. The *programming model layer* uses Kokkos to enable fast switching between various latent devices and dispatch the MPM pipeline to many major HPC platforms. Additionally, *particle/grid data, algorithms, and communication layer* with our dedicated distributed sparse grid design makes it simpler for hybrid simulation mechanics. Furthermore, we propose the dynamic load balancing algorithm to improve overall performance. Multiple experiments and comparisons are conducted to demonstrate the effectiveness and serve as a reference for simulation setups. Finally, we demonstrated that the proposed distributed MPM system can handle extremely large-scale simulations of complex elastoplastic materials with more than 1 billion particles, which has never been achieved before in computer graphics or computational mechanics.

Limitations and Future Work. First, there is still space to improve communication efficiency. We adopt MPI_Isend/MPI_Irecv for data transmission among MPI ranks. However, better communication scaling could be achieved if more efficient MPI interfaces are studied and explored. Second, as discussed in Section 7, the initial settings of MPI topology and dynamic load balancing frequency will strongly influence the overall performance. Making this decision process automatic according to initial particle samples is a valuable direction to explore. Furthermore, the workload computation in dynamic load balancing can be improved. Grid or particle alone are both insufficient to form a perfect workload description to fit all general simulation scenes. Therefore, combing these two pieces of information and conducting a better representation could provide more benefits. Finally, it is also meaningful to further improve the performance for individual parallel backends and within the *particle/grid* layers on specific devices to support faster application systems. Indeed, because the simulation framework is built on Kokkos and Cabana, testing and leveraging additional hardware architectures is straightforward. This includes AMD GPUs as deployed in the recent Frontier supercomputer, as well as future systems.

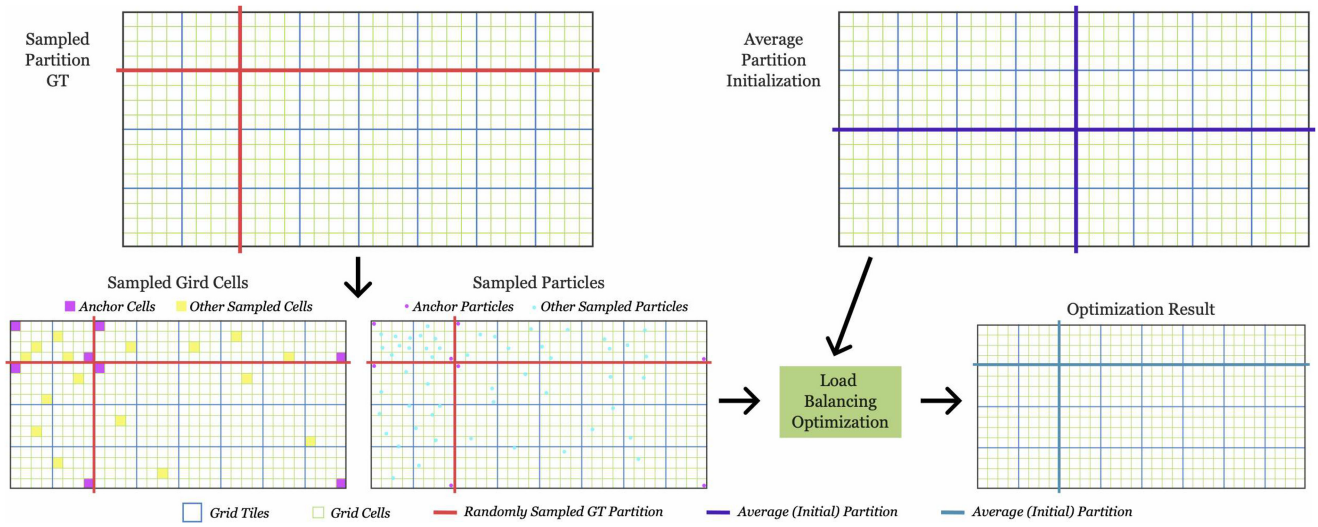

Fig. A.1. **Load balancing unit test.** In this 2D example, we sample six *grid cells* in each rank for *dynamic grid partition* test, and we sample 15 particles for *dynamic particle partition*. Each case will contain two fixed anchor *grid cells* or particles to ensure the uniqueness of the partition results.

ALGORITHM 2: Compute Workload in a Given Sub-Domain

Input: Dimension label d_i and *tile* range l_i, h_i
Input: Dimension label d_j and *tile* range l_j, h_j
Input: Dimension label d_h and *tile* range l_h, h_h
Input: Workload prefix sum matrix WS
Output: Workload in *tile* range $[l_i, h_i] \times [l_j, h_j] \times [l_k, h_k]$

```
function ComputeWorkload(d_i, l_i, h_i, d_j, l_j, h_j, d_k, l_k, h_k)
    s[d_i] ← l_i, s[d_j] ← l_j, s[d_k] ← l_k
    e[d_i] ← h_i, e[d_j] ← h_j, e[d_k] ← h_k
    return WS(e[0], e[1], e[2]) − WS(s[0], e[1], e[2])
        −WS(e[0], s[1], e[2]) − WS(e[0], e[1], s[2])
        +WS(s[0], s[1], e[2]) + WS(e[0], s[1], s[2])
        +WS(s[0], e[1], s[2]) − WS(s[0], s[1], s[2])
end function
```

APPENDICES

A DYNAMIC LOAD BALANCING

Here, we present algorithm details and the unit tests of the proposed dynamic load balancing method. Also, a 2D example is shown in Figure A.2 to visualize the dynamic optimization process. We first compute the workload matrix locally on each rank and then use MPI_ALLReduce to get the global workload matrix. Then, we compute the prefix summation for constant-time workload computation in any given rectangle region. After that, we perform iterations of 1D rectangle partition optimization. We first fix the position of the previous partition boundary p_{i-1}, then we move current partition boundary p_i to the right until finding the optimal partition position $\underset{p_i}{argmin} \sum_{jk} |w_{jk}^{p_{i-1}:p_i} - w_{jk}^{ave}|$.

A.1 Validation

To validate the partition optimization algorithm, we design a grid/particle distribution that leads to a unique ground truth of the partition boundary set. Figure A.1 illustrates a 2D case of this unit test process. We first generate a ground truth partition by random sampling. Then, we sample the same number of valid *grid cells* or particles inside each partitioned sub-domain. To ensure the uniqueness of this partition, two anchor *grid cells* or particles are included and placed at the top-left and bottom-right corners. After that, we use the average partition for initialization and perform the dynamic load balancing algorithm. The testing results show that our partition optimization algorithm can converge to the ground truth within several (usually 1–4) optimization iterations.

A.2 Workload Computation

Once the 3D prefix summation matrix of the global workload is obtained, we can calculate the workload in any given rectangle domain within constant time as shown in Algorithm 2.

A.3 Partition Optimization

We apply three separate 1D optimizations to approximate the optimal solution of the 3D partition optimization. Details of the 1D case is shown in Algorithm 3.

B RESULTS AND EVALUATIONS

The scalability tests on local workstations with OpenMP as latent programming model are shown in Figures B.4 and B.5. The scene setups are the same as the CUDA cases except for the size of the elastic cuboid. The *fixed-corotated* constitutive model [Stomakhin et al. 2012] and DGP (per 200 simulation steps) are adopted for all scaling tests.

We also test the weak scalability with more cuboids on the Summit supercomputer, which contains six NVIDIA Tesla V100 GPUs per compute node. One Summit node has two sockets, each containing three GPUs or three MPI ranks. Thus, various levels of latencies are incurred for both cross- and within-node communications. The results are shown in Figure B.6.

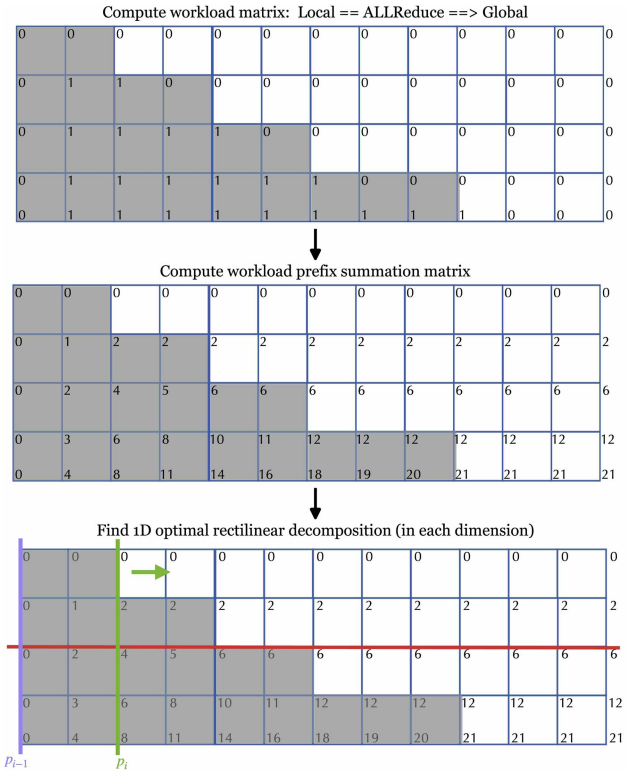

Fig. A.2. **Illustration of the dynamic load balancing algorithm.** In this 2D example, we visualize how the dynamic partition is performed on the entire simulation domain. Here, we use the grid as the workload unit for illustration, where gray grids refer to activated grid nodes, and the numbers represent the element values of the matrix.

B.1 Large-Scale Simulations

In this section, we show detailed timing statistics for large-scale simulations in Figures B.1–B.3. The high-resolution sand injection has similar detailed timing proportions as the low-res version and is not repeatedly illustrated.

C APPLICATION IMPLEMENTATION

Here, we show an example of setting up a scenario with collision objects in the *application level*.

```
// define latent programming model
using EXECSPACE = Kokkos::Cuda;
using MEMSPACE = Kokkos::CudaSpace;
using DEVICE = Kokkos::Device<Kokkos::Cuda, Kokkos::CudaSpace>;
// data type
using T = float;
// define particles
using particle_members =
    Cabana::MemberTypes<T, T[3], T[3], T[3][3], T[3][3], T>;
using particle_list = Cabana::AoSoA<particle_members, MEMSPACE>;
using particle_type = typename particle_list::tuple_type;
struct particle_index
{
    enum Names
    {
        mass = 0,
        pos = 1,
        vel = 2,
        F = 3,
        affine = 4,
        logJp = 5,
        total = 6,
    };
};
// basic parameter settings
namespace Settings
{
// domain corners
static constexpr T low_x = 0.0;
static constexpr T low_y = 0.0;
static constexpr T low_z = 0.0;
static constexpr T high_x = 200.0;
static constexpr T high_y = 200.0;
static constexpr T high_z = 200.0;
// spatial resolutions
static constexpr int res_x = 512;
```

ALGORITHM 3: 1D Rectangle Partition Optimization

Input: Dimension-of-interest: d
Output: Optimized partition: P, with $P_0 = I, P_1 = J, P_2 = K$
Output: If partition is updated: *is_changed*

function OPTIMIZATION1D(d)
▷ solve 1D rectangle optimization given partitions in the other two dimensions fixed
$d_i \leftarrow d$
$d_j \leftarrow (d+1) \mod 3$
$d_k \leftarrow (d+2) \mod 3$
for all $j \in [0, N_j), k \in [0, N_k)$ **do** ▷ in parallel
$W_{all}(j,k) \leftarrow$ COMPUTEWORKLOAD($d_i, 0, N_i, d_j, P_{d_j}(j), P_{d_j}(j+1), d_k, P_{d_k}(k), P_{d_k}(k+1)$)
$W_{ave}(j,k) \leftarrow W_{all}(i,j) \,/\, N_i$
end for
▷ N_i, N_j and N_k: rank number in corresponding dimensions
$p_{i-1} \leftarrow 0$
$p_i \leftarrow 1$
$eq_{start} \leftarrow 1$ ▷ record equivalent partition range
$last\_diff \leftarrow INT\_MAX$
$P_{d_i}(0) = 0$
for all $rank = 1, \ldots, N_i - 1$ **do**
while *true* **do**
for all $j \in [0, N_j), k \in [0, N_k)$ **do** ▷ in parallel
$W(j,k) \leftarrow$ COMPUTEWORKLOAD($d_i, p_{k-1}, p_k, d_j, P_{d_j}(j), P_{d_j}(j+1), d_k, P_{d_k}(k), P_{d_k}(k+1)$)
end for
$diff \leftarrow \sum_{j,k} |W(j,k) - W_{ave}(j,k)|$ ▷ parallel reduce
if $diff < last\_diff$ **then**
$eq_{start} \leftarrow p_k$
$last\_diff \leftarrow diff$
else
if $P(d_i, rank) \neq (p_i - 1 + eq_{start})/2$ **then**
$P(d_i, rank) \leftarrow (p_i - 1 + eq_{start})/2$
$is\_changed = true$
end if
$p_{i-1} \leftarrow p_i$
break while loop
end if
$p_i \leftarrow p_i + 1$
end while
end for
end function

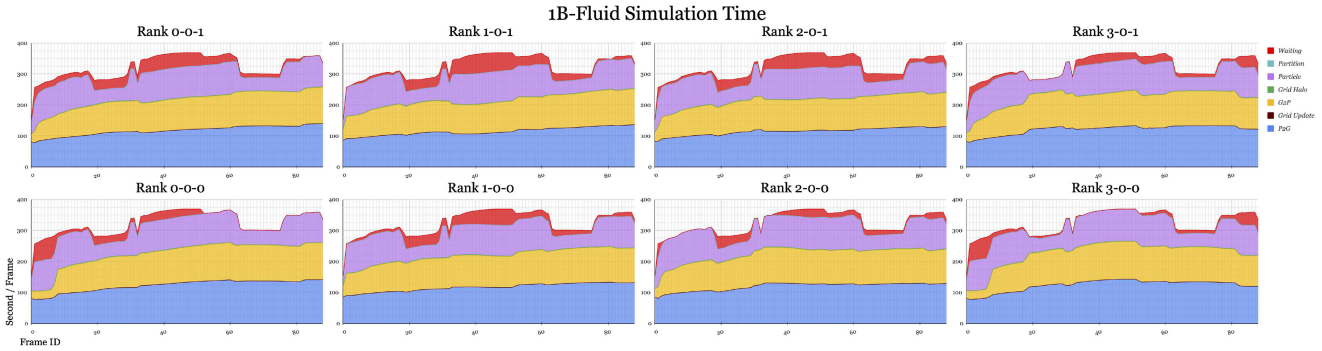

Fig. B.1. **1B-Fluid.** Detailed timing per frame (in seconds).

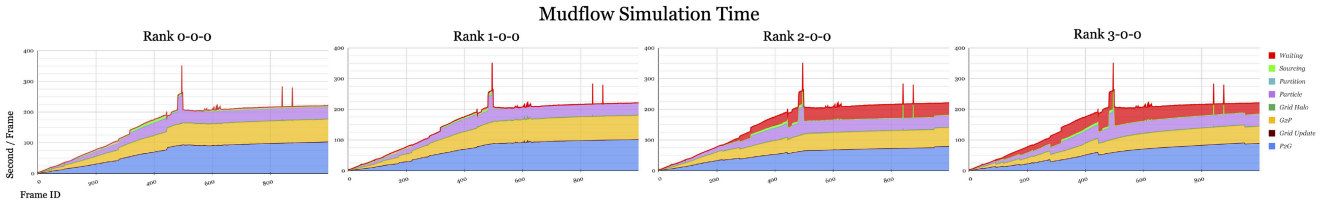

Fig. B.2. **Mudflow.** Detailed timing per frame (in seconds). Some Ethernet instability happened during particle communications in frames 495, 840, and 875, which can be regarded as external noises.

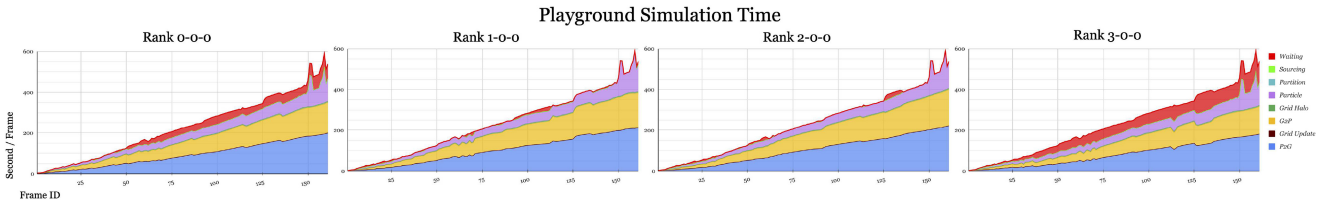

Fig. B.3. **Playground.** Detailed timing per frame (in seconds).

```
37    static constexpr int res_y = 512;
38    static constexpr int res_z = 512;
39    static constexpr T dx = high_x / res_x;
40    // halo and partition control
41    static constexpr int halo_size = 4;
42    static constexpr int num_step_rebalance = 200;
43    static constexpr bool partition_op_on = true;
44    // boundary type
45    static constexpr MultiSim::BCTypes boundary_type = MultiSim::
          BCTypes::STICKY;
46    // temporal settings
47    static constexpr int frame_num = 170;
48    static constexpr T cfl = 0.3;
49    static constexpr T fps = 4;
50    // material related settings
51    static constexpr MultiSim::MaterialTypes material_type =
52        MultiSim::MaterialTypes::FIX_COROTATED;
53    static constexpr T par_density = 1000.;
54    static constexpr T E = 9e6;
55    static constexpr T PR = 0.4;
56    // physical parameters
57    static constexpr T gravity = -9.8;
58    // particle info
59    static constexpr int PPC = 8;
60    static constexpr T par_volume =
61        ( high_x / res_x ) * ( high_y / res_y ) * ( high_z / res_z ) /
          PPC;
62    static constexpr T par_mass = par_density * par_volume;
63    static constexpr T init_y_vel = 100;
64    // sourcing related info
65    static constexpr T source_init_vel = 50;
66    // static constexpr T source_init_vel = 13;
```

```
67    // static constexpr int source_step = 800;
68    static constexpr int source_step_0 = 1;
69    // static constexpr int source_step_1 = 297;
70    static constexpr int source_step_1 = 98;
71    static constexpr int source_step_2 = 199;
72    }; // end namespace Settings
73
74    // customized particle initialization
75    template <typename Scalar, class ExecSpace>
76    struct InitParticleFunc
77    {
78        using execution_space = ExecSpace;
79        using memory_spcae = MEMSPACE;
80
81        template <class ParticleList>
82        int operator()( ParticleList& particles, const Kokkos::Array<
              Scalar, g_dim>& local_low_corner, const Kokkos::Array<Scalar,
              g_dim>& local_high_corner, const Scalar cell_size, const int
              ppc )
83        {
84            // sample from Analytical Level Set
85            MultiSim::Analytic_Shape::AnalyticLevelSet<MultiSim::
                  Analytic_Shape::cuboid, Scalar, 3>
86                cube( half_size, mid_pt, { 0, 0, 0 } );
87            std::vector<std::array<T, 3>> points;
88            MultiSim::Particle_Sample::Sampler<T, 3> sampler;
89            int particle_num = sampler.sample_particle_pos( points,
                  cube, ppc, cell_size );
90
91            // sampled particles to Kokkos:View
92            Kokkos::View<T* [3], memory_spcae> poses( "particles",
                  particle_num );
```

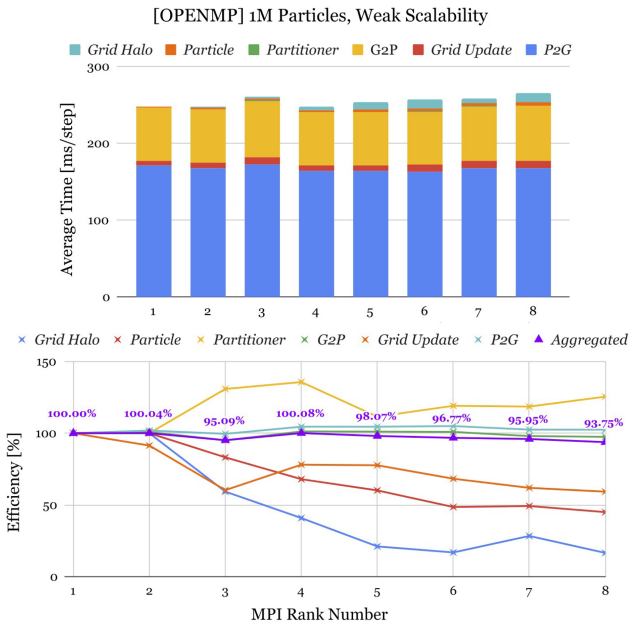

Fig. B.4. **Weak Scalability** on local workstations with CPU (OpenMP). Each rank handles an elastic box with 1M particles.

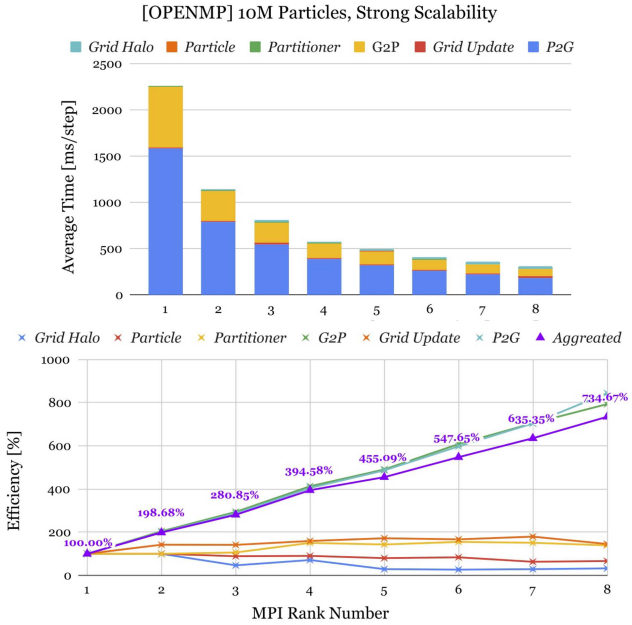

Fig. B.5. **Strong Scalability** on local workstations with CPU(OpenMP). All ranks handle a huge elastic box with 10M particles.

```
            auto host_view = Kokkos::create_mirror_view( Kokkos::
        HostSpace(), poses );
            for ( int i = 0; i < particle_num; ++i )
                for ( int d = 0; d < 3; ++d )
                    host_view( i, d ) = points[i][d];
            Kokkos::deep_copy( poses, host_view );

            // Initialize Particles
            using particle_type = typename ParticleList::tuple_type;
            using P = particle_index;
```

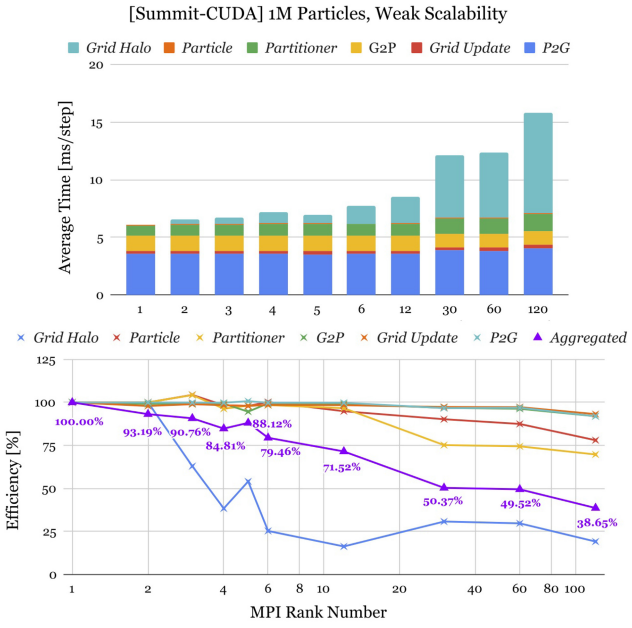

Fig. B.6. **Weak Scalability** on **Summit** with GPU(CUDA). Each rank handles an elastic box with 1M particles.

```
            particles.resize( particle_num );
            Kokkos::parallel_for( Kokkos::RangePolicy<execution_space
        >( 0, particle_num ),
                KOKKOS_LAMBDA( const int idx ) {
                    particle_type p;
                    // mass
                    Cabana::get<P::mass>( p ) = Settings::par_mass;
                    // pos
                    Cabana::get<P::pos>( p, 0 ) = poses( idx, 0 );
                    Cabana::get<P::pos>( p, 1 ) = poses( idx, 1 );
                    Cabana::get<P::pos>( p, 2 ) = poses( idx, 2 );
                    // vel
                    for ( int d = 0; d < g_dim; ++d )
                        Cabana::get<P::vel>( p, d ) = _v[d];
                    // F
                    for ( int d0 = 0; d0 < g_dim; ++d0 )
                        for ( int d1 = 0; d1 < g_dim; ++d1 )
                            Cabana::get<P::F>( p, d0, d1 ) =
                                d0 == d1 ? (Scalar)1 : (Scalar)0;
                    // C
                    for ( int d0 = 0; d0 < g_dim; ++d0 )
                        for ( int d1 = 0; d1 < g_dim; ++d1 )
                            Cabana::get<P::affine>( p, d0, d1 ) = (
        Scalar)0;
                    // logJp
                    Cabana::get<P::logJp>( p ) = 0;
                    // init particle
                    particles.setTuple( idx, p );
                } );
            Kokkos::fence();
            return particle_num;
        }
    };
    // customized scene initialization
    template <typename Scalar>
    struct InitSceneFunc
    {
        using data_type = Scalar;
        // set all
        template <class BoundaryCondition, class ProblemManager, class
        MeshType>
        void operator()( BoundaryCondition& bc, ProblemManager& pm_ptr
        , MeshType& mesh_ptr )
        {
            set_bc( bc, mesh_ptr );
            set_mat( pm_ptr );
```

```
            pm_ptr->set_gravity( Settings::gravity );
        }

    private:
        // set boundary condition - if each side has different settings
        template <class BoundaryCondition, class MeshType>
        std::enable_if_t<BoundaryCondition::bc_type == MultiSim::BCTypes::MIX, void>
        set_bc( BoundaryCondition& bc, MeshType& mesh_ptr )
        {
            using BCT = MultiSim::BCTypes;
            auto& gm = mesh_ptr->globalMeshPtr();
            std::array<BCT, g_dim * 2> bc_types;
            bc_types[0] = BCT::STICKY;
            bc_types[1] = BCT::STICKY;
            bc_types[2] = BCT::STICKY;
            bc_types[3] = BCT::STICKY;
            bc_types[4] = BCT::STICKY;
            bc_types[5] = BCT::NONE;
            bc.set_bc( bc_types[0], bc_types[1], bc_types[2], bc_types[3],
                       bc_types[4], bc_types[5], 0, 0, 0,
                       ( gm->highCorner( 0 ) - gm->lowCorner( 0 ) ) /
                           mesh_ptr->cell_size(),
                       ( gm->highCorner( 1 ) - gm->lowCorner( 1 ) ) /
                           mesh_ptr->cell_size(),
                       ( gm->highCorner( 2 ) - gm->lowCorner( 2 ) ) /
                           mesh_ptr->cell_size() );
        }
        // set boundary conditon - if all sides share the same setting
        template <class BoundaryCondition, class MeshType>
        std::enable_if_t<BoundaryCondition::bc_type != MultiSim::BCTypes::MIX, void>
        set_bc( BoundaryCondition& bc, MeshType& mesh_ptr )
        {
            auto& gm = mesh_ptr->globalMeshPtr();
            bc.set_bc( 0, 0, 0,
                       ( gm->highCorner( 0 ) - gm->lowCorner( 0 ) ) /
                           mesh_ptr->cell_size(),
                       ( gm->highCorner( 1 ) - gm->lowCorner( 1 ) ) /
                           mesh_ptr->cell_size(),
                       ( gm->highCorner( 2 ) - gm->lowCorner( 2 ) ) /
                           mesh_ptr->cell_size() );
        }
        // set material
        template <class ProblemManager>
        void set_mat( ProblemManager& pm_ptr )
        {
            auto& mat = pm_ptr->materialFunc();
            // material parameters
            mat.density = Settings::par_density;
            mat.ys = Settings::E;
            mat.pr = Settings::PR;
            mat.lambda = Settings::E * Settings::PR /
                         ( ( 1 + Settings::PR ) * ( 1 - 2 * Settings::PR ) );
            mat.mu = Settings::E / ( 2 * ( 1 + Settings::PR ) );
            mat.volume = Settings::par_volume;
        }
    };

    void test_example()
    {
        Kokkos::Array<T, g_dim * 2> global_bounding_box(
            { Settings::low_x, Settings::low_y, Settings::low_z,
              Settings::high_x,
              Settings::high_y, Settings::high_z } );
        std::array<int, g_dim> global_num_cell(
            { Settings::res_x, Settings::res_y, Settings::res_z } );
        // initializer
        InitParticleFunc<T, EXECSPACE> parpos_init_functor;
        InitSceneFunc<T> scene_init_functor;
        MultiSim::Init_Partitioner::InitUniformPartitionerFunc
            partition_init_functor;
        // solver
        auto solver = MultiSim::createMPMSolver<DEVICE, Settings::boundary_type,
                                                Settings::material_type,
                                                particle_members,
                                                particle_index>(
            global_bounding_box, global_num_cell, Settings::halo_size,
            parpos_init_functor, scene_init_functor,
            partition_init_functor,
            Settings::PPC, Settings::res_x * Settings::res_y,
            Settings::num_step_rebalance, Settings::partition_op_on,
            Settings::cfl);
        // collision object
        MultiSim::VDB_Shape::VdbLevelSet<T, 3> vdb_ls(
            INPUT_DATA_PATH, "/collision_object.vdb", { 0.0, 0.0, 0.0 } );
        MultiSim::CollisionObject<MultiSim::CollisionTypes::STICKY, T,
        g_dim, DEVICE> collision_obj( global_num_cell, Settings::dx,
        vdb_ls );
        solver->solve( Settings::frame_num, Settings::fps, std::string
        ( OUTPUT_DATA_PATH ) + "out_rank", Settings::init_y_vel,
        collision_obj, LOGGER_PATH );
    }

    int main( int argc, char* argv[] )
    {
        using T = typename Examples::T;
        MPI_Init( &argc, &argv );
        Kokkos::initialize( argc, argv );

        test_example();

        Kokkos::finalize();
        MPI_Finalize();
        return 0;
    }
```

ACKNOWLEDGMENTS

We appreciate Feng Gao's helpful advice and assistance in helping us set up the Ethernet-related hardware. We are grateful to Kayvon Fatahalian for his notes on "What Makes a (Graphics) Systems Paper Beautiful," which served as a guideline for us while preparing our system paper. We appreciate Microsoft Azure's text-to-speech technology for narrating the additional video. We would also like to express our gratitude to the anonymous reviewers for their insightful criticism.

REFERENCES

T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. 2004. Particle-based fluid simulation on GPU. In *Proceedings of the ACM Workshop on General-Purpose Computing on Graphics Processors*. Vol. 41, 42.

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

Marsha J. Berger and Shahid H. Bokhari. 1987. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers* 36, 05 (1987), 570–580.

Morten Bojsen-Hansen, Michael Bang Nielsen, Konstantinos Stamatelos, and Robert Bridson. 2021. Spatially adaptive volume tools in bifrost. In *Proceedings of the ACM SIGGRAPH 2021 Talks*. 1–2.

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 2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–11.

N. Chentanez and M. Müller. 2011. Real-time Eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics* 30, 4 (2011), 82.

N. Chentanez and M. Müller. 2013. Mass-conserving eulerian liquid simulation. *IEEE Transactions on Visualization and Computer Graphics* 20, 1 (2013), 17–29.

N. Chentanez, M. Müller, and T. Kim. 2015. Coupling 3D eulerian, heightfield and particle methods for interactive simulation of large scale liquid phenomena. *IEEE Transactions on Visualization and Computer Graphics* 21, 10 (2015), 1116–1128.

J. M. Cohen, S. Tariq, and S. Green. 2010. Interactive fluid-particle simulation using translating Eulerian grids. In *Proceedings of the 2010 Symposium on Interactive 3D Graphics and Games*. ACM, 15–22.

L. Dagum and R. Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computing in Science & Engineering* 5, 1 (1998), 46–55.

OpenBSD developers. 2021. OpenSSH. Retrieved October 20, 2022 from https://www.openssh.com/.

H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74, 12 (2014), 3202–3216.

Yu Fang, Yuanming Hu, Shi-Min Hu, and Chenfanfu Jiang. 2018. A temporally adaptive material point method with regional time stepping. *Computer Graphics Forum* 37, 8 (2018), 195–204.

Yun Fei, Yuhan Huang, and Ming Gao. 2021. Principles towards real-time simulation of material point method on modern GPUs. arXiv:2111.00699. Retrieved from https://arxiv.org/abs/2111.00699.

Ming Gao, Xinlei Wang, Kui Wu, Andre Pradhana, Eftychios Sifakis, Cem Yuksel, and Chenfanfu Jiang. 2018. GPU optimization of material point methods. *ACM Transactions on Graphics* 37, 6 (2018), 1–12.

P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola. 2010. Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, 55–64.

R. Hoetzlein. 2016. GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of the High Performance Graphics*. Eurographics Association, 109–117.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics* 38, 6 (2019), 1–16.

Yuanming Hu, Jiafeng Liu, Xuanda Yang, Mingkuan Xu, Ye Kuang, Weiwei Xu, Qiang Dai, William T. Freeman, and Fredo Durand. 2021. Quantaichi: A compiler for quantized simulations. *ACM Transactions on Graphics* 40, 4 (2021), 1–16.

Chenfanfu Jiang, Craig Schroeder, Joseph Teran, Alexey Stomakhin, and Andrew Selle. 2016. The material point method for simulating continuum materials. In *Proceedings of the ACM SIGGRAPH 2016 Courses*. 1–52.

Laxmikant V. Kale and Sanjeev Krishnan. 1993. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. 91–108.

George Karypis and Vipin Kumar. 1997. A Coarse-Grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*.

Gergely Klár, Jeff Budsberg, Matt Titus, Stephen Jones, and Ken Museth. 2017. Production ready MPM simulations. In *Proceedings of the ACM SIGGRAPH 2017 Talks*. 1–2.

G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran. 2016. Drucker-prager elastoplasticity for sand animation. *ACM Transactions on Graphics* 35, 4 (2016), 103.

Steve Lesser, Alexey Stomakhin, Gilles Daviet, Joel Wretborn, John Edholm, Noh-Hoon Lee, Eston Schweickart, Xiao Zhai, Sean Flynn, and Andrew Moffat. 2022. Loki: A unified multiphysics simulation framework for production. *ACM Transactions on Graphics* 41, 4 (2022), 1–20.

Xuan Li, Minchen Li, and Chenfanfu Jiang. 2022. Energetically consistent inelasticity for optimization time integration. *ACM Transactions on Graphics* 41, 4 (2022), 1–16.

H. Liu, Y. Hu, B. Zhu, W. Matusik, and E. Sifakis. 2018. Narrow-band topology optimization on a sparsely populated grid. In *Proceedings of the SIGGRAPH Asia 2018*. ACM, 251.

Haixiang Liu, Nathan Mitchell, Mridul Aanjaneya, and Eftychios Sifakis. 2016. A scalable schur-complement fluids solver for heterogeneous compute platforms. *ACM Transactions on Graphics* 35, 6 (2016), 1–12.

Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. 2017. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*. 513–526.

Omid Mashayekhi, Chinmayee Shah, Hang Qu, Andrew Lim, and Philip Levis. 2018. Automatically distributing eulerian and hybrid fluid simulations in the cloud. *ACM Transactions on Graphics* 37, 2 (2018), 1–14.

David S. Medina, Amik St-Cyr, and Tim Warburton. 2014. OCCA: A unified approach to multi-threading languages. arXiv:1403.0968. Retrieved from https://arxiv.org/abs/1403.0968.

OpenMPI Team Members. 2021. OpenMPI. Retrieved November 24, 2021 from https://www.open-mpi.org/

Susan M. Mniszewski, James Belak, Jean-Luc Fattebert, Christian F. A. Negre, Stuart R. Slattery, Adetokunbo A. Adedoyin, Robert F. Bird, Choongseok Chang, Guangye Chen, Stéphane Ethier, Shane Fogerty, Salman Habib, Christoph Junghans, Damien Lebrun-Grandié, Jamaludin Mohd-Yusof, Stan G. Moore, Daniel Osei-Kuffuor, Steven J. Plimpton, Adrian Pope, Samuel Temple Reeve, Lee Ricketson, Aaron Scheinberg, Amil Y. Sharma, and Michael E. Wall. 2021. Enabling particle applications for exascale computing platforms. *The International Journal of High Performance Computing Applications* 35, 6 (2021), 572–597. DOI: https://doi.org/10.1177/10943420211022829

Ken Museth. 2013. VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics* 32, 3 (2013), 1–22.

Ken Museth. 2021. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In *Proceedings of the ACM SIGGRAPH 2021 Talks*. 1–2.

T. Pfaff, N. Thuerey, J. Cohen, S. Tariq, and M. Gross. 2010. Scalable fluid simulation using anisotropic turbulence particles. *ACM Transactions on Graphics* 29, 6 (2010), 1–8.

Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. 2018. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the 13th EuroSys Conference*. 1–13.

Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. 2020. Accelerating distributed graphical fluid simulations with micro-partitioning. *Computer Graphics Forum* 39, 1 (2020), 375–388.

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics* 33, 6 (2014), 1–12.

Chinmayee Shah, David Hyde, Hang Qu, and Philip Levis. 2018. Distributing and load balancing sparse fluid simulations. *Computer Graphics Forum* 37, 8 (2018), 35–46.

Stuart Slattery, Samuel Temple Reeve, Christoph Junghans, Damien Lebrun-Grandié, Robert Bird, Guangye Chen, Shane Fogerty, Yuxing Qiu, Stephan Schulz, Aaron Scheinberg, Austin Isner, Kwitae Chong, Stan Moore, Timothy Germann, James Belak, and Susan Mniszewski. 2022. Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software* 7, 72 (2022), 4115. DOI: https://doi.org/10.21105/joss.04115

Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI–The Complete Reference: The MPI Core*. Vol. 1, MIT Press.

Alexey Stomakhin, Russell Howes, Craig Schroeder, and Joseph M. Teran. 2012. Energetically consistent invertible elasticity. In *Proceedings of the 11th ACM SIGGRAPH/Eurographics Conference on Computer Animation*. 25–32.

Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. 2013. A material point method for snow simulation. *ACM Transactions on Graphics* 32, 4 (2013), 1–10.

Igor Surmin, Alexei Bashinov, Sergey Bastrakov, Evgeny Efimenko, Arkady Gonoskov, and Iosif Meyerov. 2015. Dynamic load balancing based on rectilinear partitioning in particle-in-cell plasma simulation. In *Proceedings of the International Conference on Parallel Computing Technologies*. Springer, 107–119.

A. P. Tampubolon, T. Gast, G. Klár, C. Fu, J. Teran, C. Jiang, and K. Museth. 2017. Multi-species simulation of porous sand and water mixtures. *ACM Transactions on Graphics* 36, 4 (2017), 105.

Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (April 2022), 805–817. DOI: https://doi.org/10.1109/TPDS.2021.3097283.

O. Vantzos, S. Raz, and M. Ben-Chen. 2018. Real-time viscous thin films. *ACM Transactions on Graphics* 37, 6 (2018), 1–10.

Xinlei Wang, Yuxing Qiu, Stuart R. Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. 2020. A massively parallel and scalable multi-GPU material point method. *ACM Transactions on Graphics* 39, 4 (2020), 30–1.

T. Willhalm and N. Popovici. 2008. Putting intel® threading building blocks to work. In *Proceedings of the International Workshop on Multicore Software Engineering*. ACM, 3–4.

R. Winchenbach, H. Hochstetter, and A. Kolb. 2016. Constrained neighbor lists for SPH-based fluid simulations. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Eurographics Association, 49–56.

J. Wolper, Y. Fang, M. Li, J. Lu, M. Gao, and C. Jiang. 2019. CD-MPM: Continuum damage material point methods for dynamic fracture animation. *ACM Transactions on Graphics* 38, 4 (2019), 1–15.

K. Wu, N. Truong, C. Yuksel, and R. Hoetzlein. 2018. Fast fluid simulations with sparse volumes on the GPU. *Computer Graphics Forum* 37, 2 (2018), 157–167.

Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E. Nagel, and Michael Bussmann. 2016. Alpaka–An abstraction library for parallel kernel acceleration. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 631–640.

Received 7 August 2022; revised 7 August 2022; accepted 18 October 2022