Abstract—We propose a dynamically load-balanced parallel particle tracing algorithm, which periodically attempts to evenly redistribute particles across processes based on the k-d tree decomposition. Each process is assigned with (1) a statically partitioned, axis-aligned data block that partially overlaps with neighboring blocks in other processes and (2) a dynamically determined k-d tree leaf node that bounds the active particles for computation; the bounds of the k-d tree nodes are constrained by the geometries of data blocks. Given a certain degree of overlap between blocks, our method can balance the number of particles as much as possible. Compared with other load balancing algorithms for parallel particle tracing, the proposed method does not require any pre-analysis, does not use any heuristics based on flow features, does not make any assumptions on seed distribution, does not move any data blocks during the run, and does not need any master process for work redistribution. Based on a comprehensive performance study up to 8K processes on a Blue Gene/Q system, the proposed algorithm outperforms baseline approaches in both load balance and scalability on various flow visualization and analysis problems.

Index Terms—Parallel particle tracing, dynamic load balancing, k-d trees, performance analysis.

1 INTRODUCTION

Distributed and parallel particle tracing, which computes the movements of many massless particles that are released in the flow field, is a fundamental technique in large-scale flow visualization and analysis. Applications include visualizing streamlines and pathlines, generating streamsurfaces [9], computing finite-time Lyapunov exponents (FTLEs) and Lagrangian coherent structures (LCSs) [11], studying teleconnections [17], and analyzing differences in numerical ensembles [13]. Parallel particle tracking makes it possible to run these analyses on clusters or supercomputers and to handle large-scale data generated from computational fluid dynamics, combustion, climate, weather, and biomedical simulations.

We focus on the load balance—a known hard problem and key to achieving scalability—in parallel particle tracing. The load balancing problem exists in both task-parallel and data-parallel particle tracking methods, which are the two basic parallelism strategies. In the task-parallel methods, particles are statically distributed to parallel processes; each process has the access to the whole data. Processes are unbalanced because of the early termination of some particles that travel out of the domain or hit critical points where the velocity is zero. In the data-parallel methods, the flow data are statically partitioned into blocks and distributed to processes; particles exchange between processes to finish the tasks. Processes are unbalanced because some blocks may contain more complex features than others, such as vortices that trap particles locally.

In this paper, we propose a dynamically load-balanced parallel particle tracing algorithm using k-d (short for k-dimensional) trees. Our motivation is based on the successful use of k-d trees to balance workloads in N-body simulations [6], Delaunay tessellations [23], clustering [10], and sort-first parallel rendering [21]. In these applications, k-d trees are used to evenly (re)distribute particles, data points, or pixels across parallel processes. As illustrated in a four-process run (Figure 1(a)), each process has a k-d tree leaf node, and particles are evenly distributed into different processes periodically. However, the redistribution requires full data duplication over all parallel processes, because processes may be redistributed particles locating anywhere in the data domain, making it not scale on supercomputers.

We overcome this problem by a novel redesign of k-d tree decomposition, namely the constrained k-d tree, to redistribute particles in the data-parallel particle tracing. In this design, each process is assigned with (1) a statically partitioned, axis-aligned data block that partially overlaps with neighboring blocks in other processes and (2) a dynamically determined k-d tree leaf node that bounds the active particles for computation. The bounds of k-d tree nodes are constrained by the geometries of the data blocks. In the static data partitioning, we initially subdivide the domain into \( n \) non-overlapping, equal-sized, and axis-aligned blocks, where \( n \) equals to the number of processes. We then extrude the blocks to overlap with other blocks as much as possible, given the memory limit of the process. The extruded parts, called ghost layers, essentially maximize the overlaps between blocks under the memory limit and thus enable the k-d tree decomposition with constraints. During run time, we periodically redistribute particles based on the constrained k-d tree decomposition to balance the workload. The splitting planes in the k-d tree decomposition are limited to the overlapped regions of the ghost layers. Thus the decomposition ensures that the redistributed particles are inside the bounds of the corresponding blocks, as shown in Figure 1(b). In the extreme case that each process can fit the whole data, our algorithm can evenly redistribute the particles because there are no constraints.

The proposed method can improve load balancing in both full- and local-range flow visualization and analysis techniques for steady and unsteady flows. Full-range analyses need to trace densely seed particles over the whole domain, such as FTLE computation, while local-range analyses seed only locally in the domain, such as source-destination queries. The constrained k-d tree can be used for both static and time-varying flows in both 2D and 3D meshes.

We evaluate our method with various flow visualization and analysis tasks on Vesta, a Blue Gene/Q supercomputer at the Argonne National Laboratory. We show the performance benchmarks with up to 8K parallel processes. Compared with the baseline data-parallel particle tracing method, our constrained k-d tree approach significantly improves the performance in both load balancing and scalability. Compared with other load balancing algorithms for parallel particle tracing, the proposed method does not require any pre-analysis, does not use any heuristics based on flow features, does not make any assumptions on initial particle distribution, does not move any data blocks during the run, and does not need any master process for work redistribution.

In summary, the contributions of this paper are:

- Dynamically load-balanced parallel particle tracing algorithm based on the redesigned k-d tree decomposition that is constrained by static data partitions;
- Comprehensive performance study of our algorithm with different datasets on a Blue Gene/Q system.

In Section 2 we briefly introduce the background of this work. In Section 3, we give an overview of our method. The initialization and run-time load-balanced parallel particle tracing are described in Section 4 and Section 5, respectively. A detailed performance analysis is
we review advection-based flow visualization and analysis approaches.

We do not need any master processes to redistribute tasks. In (a) and (b), the solid lines split the particles as balanced as possible. In (b), the dashed lines show the overlapped regions of ghost layers that limit the splitting lines.

Fig. 1. Example of particle redistribution during parallel tracing through the k-d tree decomposition (a) and our constrained k-d tree decomposition (b). In (a) and (b), the solid lines split the particles as balanced as possible. In (b), the dashed lines show the overlapped regions of ghost layers that limit the splitting lines.

given in Section 6 to demonstrate the effectiveness of our method, followed by discussion in Section 7. In Section 8, we conclude our paper and discuss future work.

2 RELATED WORK

In this section, we review related work on load balancing in parallel particle tracing and on load balancing algorithms based on k-d trees. We also review advection-based flow visualization and analysis applications.

2.1 Load Balancing in Parallel Particle Tracing

Load-balanced parallel particle tracing in large-scale flow data is difficult because of the varying computation and communication workloads in both task- and data-parallel methods.

In task-parallel methods, particles are statically distributed to processes. The workload among the processes is unbalanced because the particles have different trajectories. Work stealing and work request get [8,24] can be used to redistribute tasks among processes. However, it requires a central master process for scheduling, which becomes a bottleneck to scale up. Moreover, task-parallel methods usually have higher I/O cost than data-parallel methods, because processes need to load data out-of-core when the data is larger than the memory in each process. Our method is instead based on data-parallel methods, and we do not need any master processes to redistribute tasks.

In data-parallel methods, the input data is partitioned and distributed to processes statically; particles are exchanged between processes to finish the tasks. Both static and dynamic load balancing algorithms are proposed for data-parallel particle tracing. Static load balancing algorithms usually require data preprocessing and flow feature analysis. Nouanesengy et al. [26] proposed a method to construct flow graphs that characterize both workload and flow directions of the data in preprocessing and then statically distributes the data blocks based on the optimized partitioning of flow graphs. Chen and Fujishiro [7] instead partitioned the flow data in irregular shapes based on flow feature extraction. The irregular partitions can also be achieved by hierarchical clustering [32]. Dynamic load balancing algorithms redistribute and move data blocks periodically during the run. The major costs are twofold: from the scheduling algorithm that decide how blocks should be redistributed and from the data movement. For example, Peterka et al. [28] used a recursive coordinate bisection to partition the distribution of data blocks. The repartitioning is performed periodically, and data blocks are moved on the fly. Lu et al. [19] proposed a work-stealing approach to achieve dynamic load balancing in stream-surface generation. Data blocks are moved to the processes that have lower workload.

To improve scalability, researchers have proposed hybrid methods that combine task- and data-parallelism, but the load balancing issues are not especially studied in these methods. For example, Kendall et al. [17] proposed the DStep framework that schedules tasks in a multitiered manner based on static data partitioning. DStep has also been extended to visualize ensemble differences by pathline advection analysis [13] and to analyze multivariate unsteady flow data with Lagrangian-based attribute space projection [12]. Zhang et al. [34] also modify the DStep framework to generate high-order access dependencies in a data preprocessing stage.

We regard our load-balanced particle tracing algorithm as a hybrid method that distributes data blocks statically and redistributes particles dynamically. Comprehensive comparisons with previous methods are discussed in Section 7.

2.2 Load Balancing based on K-D Trees

A comprehensive review of load balancing for distributed and parallel computing in general is out of the scope of this paper. We instead focus on load balancing algorithms that are based on k-d trees.

The k-d tree, which was invented by Bentley [4], is a data structure that splits k-dimensional data for efficient range queries and k-neighbor queries. The k-d tree data structure is essentially a balanced binary tree; each leaf node of the tree is a subdomain of its parent.

K-d trees have been used in load balancing of various applications in computational sciences, database, data analysis, and visualization. In computational sciences, k-d trees are used to build hierarchies of particles in order to improve scalability in large N-body cosmology simulations [3,6] and molecular dynamics simulations [30]. In database applications, k-d trees can be used for fast query of images [1]. In data analysis, k-d trees are widely used in distributed and parallel clustering algorithms, such as k-means [10] and the DBSCAN algorithm [27]. In visualization, the community has been extensively using k-d trees to address various load-balancing problems. For ex-
ample, Morozov and Peterka [23] proposed a load-balanced Delaunay tessellation algorithm. Input points are partitioned with k-d trees and distributed to different processes for load-balanced computation. In sort-first volume rendering, k-d trees are used to split the image space to balance the parallel rendering costs [21].

The implementation of k-d trees must be efficient and scalable for distributed applications. Serial k-d tree implementations include nanoflann1, but they are not able to scale in parallel. Zhang et al. [33] proposed the first scheme to distribute k-d trees in peer-to-peer systems. Morozov and Peterka [23] further improved the algorithm to scale the k-d tree decomposition for efficient Delaunay tessellation.

In our study, we redesigned the parallel tree construction algorithm for constrained k-d trees. We further use the constrained k-d trees to balance the number of particles in parallel processes to achieve dynamic load balancing.

2.3 Advection-Based Flow Visualization and Analysis

We review advection-based flow visualization and analysis applications that can benefit from scalable and parallel particle tracing. Advection-based flow visualization and analysis methods include texture-based [18], geometry-based [20], and part of feature extraction and tracking techniques [29]. Based on the distribution of particle seed locations, we subdivide the techniques into two categories: full-range analysis and local-range analysis.

In local-range analyses, particles are seeded in sparse and local regions in order to understand part of the data. For example, seeds of a stream surface are usually distributed on a line or curve in 3D flows [9]. A source-destination query [17] needs to seed particles only in a local region, instead of the whole domain.

In full-range analyses, particle are seeded densely in the entire data domain in order to analyze and visualize comprehensive features in the flow. For example in texture-based of visualization, line integral convolution (LIC) [5] and unsteady flow LIC (UFLIC) [31] needs to compute streamlines and pathlines, respectively, from all locations. Although methods have been proposed to reduce the computational cost of tracing densely seeded particles, such as partial path reuse [15] and adaptive refinement [2], comprehensive analysis of the computation of FTLEs and LCSs [14] still needs scalable and parallel tracing of densely seeded particles [25].

Our study can be used to improve load balancing in both full- and local-range analyses. Our method is also compatible with both static and time-varying flows. More details and evaluations are in the following sections.

3 Overview

As described in Algorithm 1 and illustrated in Figure 2, the pipeline of our method consists of a initialization stage and a computation stage.

The initialization stage has three steps. First, we partition the input data into non-overlapping, equal-sized, axis-aligned blocks. The number of blocks is equal to the number of parallel processes, and each process has single block. Second, we extrude the ghost layers of each block to overlap with other blocks as much as possible, given the memory limit of each process. The ghost layers enable the k-d tree decomposition to place splitting planes in the overlapped regions.

Third, we load the blocks into memory with parallel I/O. The input particles are also loaded and initialized in the corresponding blocks that the particles sit in. More details are explained in Section 4.

The computation stage is an iterative process that alternatively executes the particle redistribution and particle tracing. In the particle redistribution phase, the parallel processes collectively run the constrained k-d tree decomposition, which simultaneously exchanges unfinished particles to the destination processes to continue the computation. In the particle tracing phase, each parallel process independently traces the unfinished particles in this process without any interprocess communication. More details are given in Section 5.

4 Algorithm Initialization

We initialize data blocks and particles for the dynamical load-balanced parallel particle tracing.

Domain partitioning We partition the input data by iteratively splitting the domain in each dimension; the order of dimensions is consistent with that of the k-d tree decomposition. Without loss of generality, we assume the number of processes $n$ is a power of two;

---

1https://github.com/jblancoc/nanoflann

---
otherwise we can subdivide each dimension based on the prime factorization of the number. For example, in 3D data we evenly subdivide the domain along the x-axis into two blocks and then further evenly split each block along the y and z axes. After the z-axis, the next dimension to subdivide is the x-axis. The iteration stops when the number of blocks equals \( n \). Each process then owns single block. The outputs of the domain partitioning are \( n \) non-overlapping, equal-sized, axis-aligned data blocks.

**Block extrusion** We maximize the overlaps between blocks by adding ghost layers on these blocks. As shown in Figure 3, each block is extruded in all dimensions so that the block overlaps with its neighbor blocks. The block extrusion enables the k-d tree to split the domain in the overlapped regions, as explained in the next section. The thickness of the block is bounded by the available memory in the process. In the extreme case when the memory is large enough to fit the whole dataset, each process keeps a complete copy of the data.

**Block I/O** We load the extruded data blocks in parallel. The parallel I/O is handled by the block I/O layer (BIL) [16], which essentially uses parallel I/O to load extruded data blocks in a scalable manner.

**Particle initialization** We load input particles and distribute them into different processes. Each particle is assigned to the block whose “core” region excluding ghost layers contains the particle.

## 5 Load-Balanced Parallel Particle Tracing

The computation stage (the loop in Algorithm 1) of our algorithm alternates the collective particle redistribution and the serial particle tracing. The collective particle redistribution exchanges unfinished particles based on the constrained k-d tree decomposition; the serial particle tracing locally computes the movements of unfinished particles in each process without communication.

### 5.1 Particle Redistribution based on Constrained K-d Trees

We redesign the distributed k-d tree decomposition algorithm in by constraining the splitting planes in the overlapped regions of data blocks. Particles are also redistributed to parallel processes during the k-d tree construction process.

The constrained k-d tree construction is illustrated in Figures 5 and 4. The algorithm cycles through each dimension to split the data domain. For each dimension, the algorithm attempts to find an axis-aligned median plane that can split particles evenly. If the plane is not in the overlapped regions between the two neighbor blocks, we use the boundary of either block that can balance the number of particles. As shown in the 2D example in Figure 5, if the coordinate of the plane is greater than the upper bound of the red block in \( x \) direction, we split the dimension with the left boundary of the blue block; otherwise we choose the median plane in the overlap region of the two blocks. The rule is similar in 3D, as illustrated in Figure 4.

We redesign the distributed k-d tree decomposition [23] to constrain the splitting planes, as detailed in Algorithm 2. Each iteration has two key parts: median selection and particle exchange. Initially, we group all processes in a group. In the first iteration, each process sends all its particles to a designated root process within the group. The root process then picks up a median plane that is constrained in the overlapped region of ghost layers, as detailed in Algorithm 2. Each iteration has two key parts: median selection and particle exchange. Initially, we group all processes in a group. In the first iteration, each process sends all its particles to a designated root process within the group. The root process then picks up a median plane that is constrained in the overlapped regions as described above, and broadcasts it to all other processes within the group. The selected median value splits all the particles into two parts, trying to make each part have particles as even as possible under the geometry constraints. For particle exchange, the processes are divided into two subgroups. Each process from one subgroup chooses a partner from the other subgroup to exchange particles, so that half of the processes receive the particles whose projection onto the first coordinate is less than the picked median value, while other half receives particles whose projection is more than the value. Given \( n \) processes, the algorithm will repeat in each process subgroup by cycling through the coordinates for splitting until \( \log n \) iterations are executed.

The constrained k-d tree decomposition splits the particles as evenly as possible while also ensuring the particles are bounded in the data block of corresponding process. A thicker ghost layer leads to more even decomposition of the number of particles and better workload balance.

### 5.2 Particle Tracing within Processes

We trace particles inside the local block of each process independently without communication, as shown in Algorithm 3. After tracing, each particle is either finished or unfinished. A particle is marked as finished if it goes out of the data domain, or hits a critical point where the velocity is zero, or reaches the maximum integral steps required by the visualization and analysis algorithm; otherwise the particle is

---

**Algorithm 2** redistribute\_particles(comm, local\_particles)

```
\begin{algorithm}
\begin{algorithmic}
  \State \textbf{dim} \gets 0
  \State \textbf{rank} \gets \text{comm.rank}
  \State \textbf{domain} \gets \text{data.domain}
  \State \textbf{group} \gets \{0, 1, \ldots, n-1\}
  \State \textbf{N_r} \gets \log n \quad \triangleright \text{number of rounds}
  \For {i \gets \text{root}[0]} \quad \triangleright \text{rank of the root process in the group}
    \State \textbf{particles} \gets \text{comm.gather(group, root, local\_particles)} \quad \triangleright \text{gather to the root}
    \If{\text{rank} = \text{root}}
      \State range \gets \text{compute\_overlapped\_region(domain, dim)} \quad \triangleright \text{overlapped region of ghost layers}
      \State median \gets \text{compute\_bounded\_median(particles, range, dim)}
    \EndIf
    \State median \gets \text{comm.broadcast(group, root, median)} \quad \triangleright \text{broadcast from the root}
    \State partner \gets (\text{rank} + \text{group.size}/2) \mod \text{group.size}
    \State \textbf{particles\_low} \gets \{ p \in \text{local\_particles} \mid p[\text{dim}] < \text{median} \}
    \State \textbf{particles\_up} \gets \{ p \in \text{local\_particles} \mid p[\text{dim}] \geq \text{median} \}
    \If{\text{rank} < \text{group.size}/2}
      \State local\_particles \gets \text{particles\_low}
      \State group \gets \{ x \in \text{group} \mid x < \text{group.size}/2 \}
    \Else
      \State local\_particles \gets \text{particles\_up}
      \State group \gets \{ x \in \text{group} \mid x \geq \text{group.size}/2 \}
    \EndIf
    \State domain \gets \text{bisect\_domain(domain, median, dim)}
    \State \textbf{dim} \gets (\text{dim} + 1) \mod k \quad \triangleright k \text{ is the dimensionality of k-d tree}
    \State i \gets i + 1
  \EndFor
\end{algorithmic}
\end{algorithm}
```
We conducted a comprehensive performance study with three applications: tracing densely seeded streamlines in the thermal hydraulics simulation (Nek5000) data, querying source-destinations in the GEOS-5 simulation data, and computing FTLE of Hurricane Isabel data. The detailed specification of the data are detailed in Table 1 and the following subsections.

The implementation of our method is based on C++11. We use the DIY2 library [22], which wraps MPI for interprocess communication, to decompose the domain and exchange messages between processes. We also use the BIL library [16] to efficiently read disjoint data blocks from different NetCDF files collectively.

The benchmark platform is Vesta, an IBM Blue Gene/Q system at Argonne National Laboratory. Vesta has 2,048 compute nodes: each has a 16-core 1.6 GHz PowerPC A2 processor and 16 GB of DDR3 RAM. The interconnection between compute nodes is a 5D torus network. The storage of Vesta is GPFS (IBM General Parallel File System). In our test, we execute 8 processes per compute node, and we use up to 8,192 processes. We study the performance of our method with the following benchmarks.

- **Load balancing** We use the maximize workload subdivided by the average workload as the load balancing indicator. The workload in each process is evaluated as the number of numerical integral steps for particle tracing. We also use a Gantt chart to visualize the computation time of different processes.
  - **Strong scaling** We measure how the total execution time varies with the number of processes for a fixed total number of particles.
  - **Weak scaling** We measure how the total execution time varies with the number of processes for a fixed number of particles per process.

We also alter the conditions that affect performance:
  - **Available memory (M)**. We arbitrarily choose different memory limits for data blocks per process to test the scalabilities. The thickness of the ghost layers grows with the memory limit;
  - **Choice of 3D/4D trees (k)**. We study whether 3D or 4D trees perform better in time-varying flows.
  - **Maximum integral steps per round (N_{max})**. We study the only parameter that controls the frequency of particle redistribution.

The baseline approach in the performance study is a data-parallel particle tracing implementation. The initialization stage of the baseline approach is the same as in Section 4. The computation stage alternates the independent particle tracing and the collective particle exchange phases. In the first phase, each particle does not stop until it goes out of the local block; in the second phase, particles that move to remote processes are exchanged collectively with MPI_Alltoall. The program exits after all tasks in the system are finished.

### 6 Performance Analysis

We conducted a comprehensive performance study with three applications: tracing densely seeded streamlines in the thermal hydraulics simulation (Nek5000) data, querying source-destinations in the GEOS-5 simulation data, and computing FTLE of Hurricane Isabel data. The detailed specification of the data are detailed in Table 1 and the following subsections.

The implementation of our method is based on C++11. We use the DIY2 library [22], which wraps MPI for interprocess communication, to decompose the domain and exchange messages between processes. We also use the BIL library [16] to efficiently read disjoint data blocks from different NetCDF files collectively.

The benchmark platform is Vesta, an IBM Blue Gene/Q system at Argonne National Laboratory. Vesta has 2,048 compute nodes: each has a 16-core 1.6 GHz PowerPC A2 processor and 16 GB of DDR3 RAM. The interconnection between compute nodes is a 5D torus network. The storage of Vesta is GPFS (IBM General Parallel File System). In our test, we execute 8 processes per compute node, and we use up to 8,192 processes. We study the performance of our method with the following benchmarks.

- **Load balancing** We use the maximize workload subdivided by the average workload as the load balancing indicator. The workload in each process is evaluated as the number of numerical integral steps for particle tracing. We also use a Gantt chart to visualize the computation time of different processes.
  - **Strong scaling** We measure how the total execution time varies with the number of processes for a fixed total number of particles.
  - **Weak scaling** We measure how the total execution time varies with the number of processes for a fixed number of particles per process.

We also alter the conditions that affect performance:
  - **Available memory (M)**. We arbitrarily choose different memory limits for data blocks per process to test the scalabilities. The thickness of the ghost layers grows with the memory limit;
  - **Choice of 3D/4D trees (k)**. We study whether 3D or 4D trees perform better in time-varying flows.
  - **Maximum integral steps per round (N_{max})**. We study the only parameter that controls the frequency of particle redistribution.

The baseline approach in the performance study is a data-parallel particle tracing implementation. The initialization stage of the baseline approach is the same as in Section 4. The computation stage alternates the independent particle tracing and the collective particle exchange phases. In the first phase, each particle does not stop until it goes out of the local block; in the second phase, particles that move to remote processes are exchanged collectively with MPI_Alltoall. The program exits after all tasks in the system are finished.

#### 6.1 Thermal Hydraulics Data

The thermal hydraulics dataset is from the output of the Nek5000 solver, which is a large-eddy Navier-Stokes simulation developed by Argonne National Laboratory. We use single timestep of the data for the static flow analysis, and trace streamlines from all grid points. Figure 6 show the Gantt chart visualization of a run, and Figure 7 shows the timings of both strong and weak scaling results with different memory limits.

**Load balance** The workload in our method is much more balanced than in the baseline approach. We collect and visualize in Figure 6 the performance data from two 64-process runs with our method and the baseline method, respectively. We use the down-sampled version of the original data in order to fit into the small number of processes (64) for Gantt chart visualization. In the Gantt chart, the horizontal axis is the execution time, and the vertical axis is the ID of the process. We also visualize the load balance indicator over the execution time. We can see that the particle redistribution time is much less than the particle tracing time.

**Strong scaling** We trace 16 million particles in total with 512, 1K, 2K, 4K, and 8K processes to study the strong scalability. As shown in Figure 7(a), as the number of processes increases, the total execution time of our method decreases faster than the baseline approach does. The strong scalability of our method is better than the baseline. With 8K processes, our method performs more than twice as fast as the baseline. The parallel efficiency of 2K processes with a 48 MB, 96 MB, and 384 MB memory limit is 44.9%, 47.7%, 68.5%, respectively.
### 6.2 GEOS-5 Simulation Data

The GEOS-5 data is from the simulation output of an atmospheric model developed by the NASA Goddard Space Flight Center. The spatial resolution of the model is a 1° x 1.25° latitude-longitude grid with 72 vertical pressure levels that ranges from 1 atm (near the terrain surface) to 0.01 hPa (about 80 km). The dataset in our experiment consists of the monthly averaged output from January 2000 to December 2001.

We conduct a source-destination query analysis with 8M particles in the GEOS-5 data. The query results are shown in Figure 8(a). In this experiment, seeds are densely placed in North America, and we visualize the distribution of the particle trajectories over time.

**Strong scaling** The first row of Figure 9(a)(b)(c) shows the strong scaling tests of this case. The optimal scaling curves in this figure shows the linear speedup for reference. Figure 9(a) shows that our method scales better with larger memory limits. Although larger memory limits lead to thicker ghost layers and eventually result in higher I/O costs, I/O takes less than 10% of the total execution time in all cases. Larger memory limits improve both strong and weak scalabilities, because thicker ghost layers weaken the constraints in our k-d tree decomposition.

### 6.3 Hurricane Isabel Data

The Isabel dataset is from an atmospheric simulation conducted by the National Center for Atmospheric Research. The resolution of this mesh is 500 x 500 x 100, and the dataset has 48 time steps. We compute the FTLE field (shown in Figure 8(b)) based on the wind (U, V, and W components) by tracing 12 million particles that are uniformly seeded in the domain. The optimal $N_{\text{max}}$ in this experiment is 50.

**Strong scaling** Our method greatly improves the performance and load balance in the strong scaling test, as shown in the first row of Figure 10. The computation is 2.1 times compared with the baseline method using 8K processes. We can see that the larger memory limit leads to higher I/O cost because the extruded blocks are larger, but our method scales better with larger memory limit in general.

### 7 Discussions

In this section, we compare our method with other load balancing methods and discuss the limitations of our method.

#### 7.1 Comparison with Other Methods

We compare our method with other load balancing algorithms for parallel particle tracing in different aspects, which are detailed in Table 2 and below.

**Data preprocessing** Our method does not require any data preprocessing. In previous studies, such as flow graph based workload estimation [26] and irregular data partitioning [7, 32], the whole input dataset need to be processed and analyzed in a pilot run before the actual particle tracing runs. As the data scale continues to grow in present and future computing, however, it becomes prohibitive to perform the preprocessing because of high I/O costs.

**Feature analysis** Our method does not rely on any flow feature analysis. Flow partitioning [7] and clustering [32] can be expensive and sensitive to parameter selection. It is still an open challenge to well define features in fluid flows, especially time-varying datasets. It is even harder to detect features in distributed environments with scalability. In contrast, our constrained k-d tree method is light-weighted and scalable.

**Seed distribution** We make no assumptions about the initial particle distribution. Most load balancing algorithms in data-parallel particle tracing [7, 26, 32] are designed for full-range analyses that trace densely seeded particles in the whole domain. Our method can handle arbitrary distributed seeds for load balanced particle tracing.

**Data dimensionality** Our method supports both static and time-varying data in both 2D and 3D meshes, because k-d tree can decompose spaces in arbitrary dimensions. To the best of our knowledge, most load balancing algorithms in data-parallel settings are designed for static datasets. Task-parallel algorithms can support unsteady flows, but they require out-of-core I/O scheduling for large-scale data.

**Data movement** The data movement of our method is minimal. Unlike previous studies that move data blocks for dynamic load balancing [19, 28], we do not need to move flow data back and forth after the initialization. Moving data blocks can be expensive, especially for clusters that have lower network bandwidths. The only data movement in our method is the particle exchanging, which is also necessary in all data-parallel particle tracing algorithms.

**Communication patterns** The distributed k-d tree decomposition is self-consistent and decentralized. In previous work stealing [8] and work requesting [24] algorithms for task-parallel particle tracing, a dedicated process is necessary to schedule the workloads in all other processes. The 1-to-n communication pattern makes it difficult to scale in large systems. We instead do not need any master processes to schedule tasks.

#### 7.2 Limitations

Our method does not guarantee optimal and perfect load balance. First, we assume that an equal number of particles leads to the ideal balance, but it is only an approximation. In the independent particle tracing phase, some particles may stop earlier than others, because they go

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Resolution</th>
<th>Analysis</th>
<th>Seeding</th>
<th>Benchmark</th>
<th>Alternating Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nek5000</td>
<td>1024^1</td>
<td>12 GB</td>
<td>Full-range</td>
<td>Strong/weak scaling</td>
<td>M</td>
</tr>
<tr>
<td>GEOS-5</td>
<td>288 x 181 x 72 x 24</td>
<td>1.34 GB</td>
<td>Source-destination</td>
<td>Local-range</td>
<td>Strong scaling</td>
</tr>
<tr>
<td>Isabel</td>
<td>500 x 500 x 100 x 48</td>
<td>13.4 GB</td>
<td>FTLE</td>
<td>Full-range</td>
<td>Strong scaling</td>
</tr>
</tbody>
</table>

In the baseline method, the parallel efficiency is 38.4%, 40.6%, 42.2%, respectively. We can also see that the particle tracing time dominates the execution time, even if the percentage of I/O and particle redistribution increase as the number of processes increases.

**Weak scaling** We trace 2K particles per process with 512, 1K, 2K, and 8K processes in this test. As the number of processes increases, the execution time of the baseline approach increases much faster than that our method, as shown in Figure 7(c). In other words, the weak scalability of our method is also better than that of the baseline. The parallel efficiency of 2K processes with a 48 MB, 96 MB, and 384 MB memory limit in our method is 57.5%, 59.1%, and 79.8%, respectively. In the baseline method, the corresponding parallel efficiency is 40.7%, 54.8%, 62.4%, respectively.

From all tests in Figure 7, we can see that our method performs better with larger memory limits. Although larger memory limits lead to thicker ghost layers and eventually result in higher I/O costs, I/O takes less than 10% of the total execution time in all cases. Larger memory limits improve both strong and weak scalabilities, because thicker ghost layers weaken the constraints in our k-d tree decomposition.

#### 6. Hurricane Isabel Data

The Isabel dataset is from an atmospheric simulation conducted by the National Center for Atmospheric Research. The resolution of this mesh is 500 x 500 x 100, and the dataset has 48 time steps. We compute the FTLE field (shown in Figure 8(b)) based on the wind (U, V, and W components) by tracing 12 million particles that are uniformly seeded in the domain. The optimal $N_{\text{max}}$ in this experiment is 50.

**Strong scaling** Our method greatly improves the performance and load balance in the strong scaling test, as shown in the first row of Figure 10. The computation is 2.1 times compared with the baseline method using 8K processes. We can see that the larger memory limit leads to higher I/O cost because the extruded blocks are larger, but our method scales better with larger memory limit in general.

**Choice of 3D/4D trees** We test whether 3D or 4D tree can achieve better load balance with our method. The results are shown in the second row of Figure 10. In general, we can see that the performance is better if we consider only the space dimensions in this experiment. We conclude that the workload of using 4D trees is less balanced than using 3D trees, because the time of spatiotemporal particles always increases and leads to imbalance in the time dimension.

### Data preprocessing

Our method does not require any data preprocessing. In previous studies, such as flow graph based workload estimation [26] and irregular data partitioning [7, 32], the whole input dataset need to be processed and analyzed in a pilot run before the actual particle tracing runs. As the data scale continues to grow in present and future computing, however, it becomes prohibitive to perform the preprocessing because of high I/O costs.

**Feature analysis** Our method does not rely on any flow feature analysis. Flow partitioning [7] and clustering [32] can be expensive and sensitive to parameter selection. It is still an open challenge to well define features in fluid flows, especially time-varying datasets. It is even harder to detect features in distributed environments with scalability. In contrast, our constrained k-d tree method is light-weighted and scalable.

**Seed distribution** We make no assumptions about the initial particle distribution. Most load balancing algorithms in data-parallel particle tracing [7, 26, 32] are designed for full-range analyses that trace densely seeded particles in the whole domain. Our method can handle arbitrary distributed seeds for load balanced particle tracing.

**Data dimensionality** Our method supports both static and time-varying data in both 2D and 3D meshes, because k-d tree can decompose spaces in arbitrary dimensions. To the best of our knowledge, most load balancing algorithms in data-parallel settings are designed for static datasets. Task-parallel algorithms can support unsteady flows, but they require out-of-core I/O scheduling for large-scale data.

**Data movement** The data movement of our method is minimal. Unlike previous studies that move data blocks for dynamic load balancing [19, 28], we do not need to move flow data back and forth after the initialization. Moving data blocks can be expensive, especially for clusters that have lower network bandwidths. The only data movement in our method is the particle exchanging, which is also necessary in all data-parallel particle tracing algorithms.

**Communication patterns** The distributed k-d tree decomposition is self-consistent and decentralized. In previous work stealing [8] and work requesting [24] algorithms for task-parallel particle tracing, a dedicated process is necessary to schedule the workloads in all other processes. The 1-to-n communication pattern makes it difficult to scale in large systems. We instead do not need any master processes to schedule tasks.

#### 7.2 Limitations

Our method does not guarantee optimal and perfect load balance. First, we assume that an equal number of particles leads to the ideal balance, but it is only an approximation. In the independent particle tracing phase, some particles may stop earlier than others, because they go...
Fig. 6. Performance of 64 processes tested by downsampled Nek data. Top: Gantt chart using the baseline method. Each row in the Gantt chart represents a process. Middle: Line chart showing the evolution of workload balance. The dashed line represents the baseline method, while the solid line represents the k-d tree method. Bottom left: Gantt chart using our k-d tree method. The time axes of the two Gantt charts and the line chart are aligned. Bottom right: Rendering result of the Nek data with 2,000 streamlines. The color encodes the integration steps during the streamline generation.

Table 2. Comparison with other load balancing algorithms for parallel particle tracing. The plus (+) means that the method meets a certain kind of category or requirement, while minus (-) means that it does not meet the corresponding kind of category or requirement.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel over Data</td>
<td>Data Preprocessing</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Parallel over Seeds</td>
<td>Heuristics on Flow Features</td>
<td>-</td>
<td></td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Static Load Balance</td>
<td>Assumptions on Initial Seed</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Load Balance</td>
<td>Flow Data Movement</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Scheduler Process</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>
Fig. 7. Performance breakdown for Nek data analysis. (a) and (b) show the strong scaling tests, while (c) and (d) show the weak scaling tests. At each process count, the six stacked histograms represent the baseline method with 48 MB memory, the k-d tree method with 48 MB memory, the baseline method with 96 MB memory, the k-d tree method with 96 MB memory, the baseline method with 384 MB memory, and the k-d tree method with 384 MB memory, respectively.

Out of the local bounds of the block or hit critical points. The computation time may also vary because of data locality. As a result, processes have different workloads even if they are assigned the same number of particles. Second, the constrained k-d tree decomposition does not guarantee an even distribution of particles. Because the splitting planes of k-d trees are limited to the overlapped regions of ghost layers, the k-d tree could be imbalanced.

Our method needs the parameter configuration of $N_{\text{max}}$. As the single important parameter, it determines how frequently the particles are redistributed. If $N_{\text{max}}$ is too small, the k-d tree decomposition may bring large overhead. If $N_{\text{max}}$ is too large, the workload will be uneven because the redistribution is not performed frequently enough. We have shown the optimal values in our experiments on Blue Gene/Q systems, but the number may vary with different data and different architectures.

Our method is bulk-synchronous. The constrained k-d tree decomposition is a collective operation that involves all processes in a synchronous manner. However, the cost of synchronization grows as the number of processes increases. We currently cannot overlap computation with communication.

8 Conclusions and Future Work

In this work, we present a novel approach to balance workloads in parallel particle tracing with constrained k-d trees. The static data partitioning with partial data replication makes it possible to redistribute the unfinished particles during particle tracing through the k-d tree decomposition. A balanced workload is achieved dynamically with our method, which improves the performance of parallel particle tracing. We evaluate our method with different flow visualization and analysis problems through a comprehensive performance analysis. Results show that the proposed method improves both load balance and scalability in particle tracing.

In the future, we will generalize our algorithm to handle unstructured mesh data. Our method also has the potential to visualize and analyze flows in situ. We would also like to combine our algorithm with existing static load balancing strategies. Furthermore, we plan to relax the synchronizations in the parallel k-d tree decomposition.

References

The maximal number of tracing steps, $N_{\text{max}}$, is set to 20 in (a), (b), and (c). Panels (d), (e), and (f) show the tests with different $N_{\text{max}}$ under 64 MB memory limit.

Fig. 9. Strong scaling tests using GEOS-5 data with different process counts. Panels (a), (b), and (c) show the tests with different memory limits. The maximal number of tracing steps, $N_{\text{max}}$, is set to 20 in (a), (b), and (c). Panels (d), (e), and (f) show the tests with different $N_{\text{max}}$ under 64 MB memory limit.

Fig. 10. Strong scaling tests using Isabel data with different process counts. (a), (b), and (c) show the tests with different memory limits that consider both space and time dimensions used for splitting (i.e., 4D trees), while (d), (e), and (f) show the comparison between considering both space and time dimensions for splitting (i.e., 4D trees) and considering only space dimensions for splitting (i.e., 3D trees).


