Extreme-Scale Stochastic Particle Tracing for Uncertain Unsteady Flow Analysis

Hanqi Guo
Argonne National Laboratory
9700 S. Cass Ave.,
Argonne, IL 60439 USA
Email: hguo@anl.gov

Wenbin He
The Ohio State University
2015 Neil Ave.,
Columbus, OH 43210 USA
Email: he.495@buckeyemail.osu.edu

Sangmin Seo
Argonne National Laboratory
9700 S. Cass Ave.,
Argonne, IL 60439 USA
Email: sseo@anl.gov

Han-Wei Shen
The Ohio State University
2015 Neil Ave.,
Columbus, OH 43210 USA
Email: shen.94@osu.edu

Tom Peterka
Argonne National Laboratory
9700 S. Cass Ave.,
Argonne, IL 60439 USA
Email: tpeterka@mcs.anl.gov

***Abstract*—We present an efficient and scalable solution to estimate uncertain transport behaviors using stochastic flow maps (SFM,) for visualizing and analyzing uncertain unsteady flows. SFM computation is extremely expensive because it requires many Monte Carlo runs to trace densely seeded particles in the flow. We alleviate the computational cost by decoupling the time dependencies in SFMs so that we can process adjacent time steps independently and then compose them together for longer time periods. Adaptive refinement is also used to reduce the number of runs for each location. We then parallelize over tasks—packets of particles in our design—to achieve high efficiency in MPI/thread hybrid programming. Such a task model also enables CPU/GPU coprocessing. We show the scalability on two supercomputers, Mira (up to 1M Blue Gene/Q cores) and Titan (up to 128K Opteron cores and 8K GPUs), that can trace billions of particles in seconds.**

I. Introduction

Visualizing and analyzing data with uncertainty are important in many science and engineering domains, such as climate and weather research, computational fluid dynamics, and materials science. Instead of analyzing deterministic data, scientists can gain more understanding by investigating uncertain data that are derived and quantified from experiments, interpolation, or numerical ensemble simulations. For example, typical analyses of uncertain flows involve finding possible pollution diffusion paths in environmental sciences with uncertain source-destination queries and locating uncertain material boundaries in computational fluid dynamic models with uncertain Lagrangian analysis.

In this work, we develop a scalable solution to compute *stochastic flow maps* (SFMs), which characterize transport behaviors in uncertain unsteady flows. SFMs are the generalization of flow maps of deterministic data and hence are the basis for uncertain flow analysis. Formally, the flow map is a function that maps the start location and the end location after time T in a flow field; the SFM follows the same definition except that the end location is stochastic. Applications based on SFMs include not only uncertain source-destination queries but also uncertain flow separatrix extraction [1] and uncertain flow topology analysis [2]. For example, finite-time Lyapunov exponent (FTLE) analysis can be generalized to understand uncertain transport behaviors in uncertain flows [1]. The distribution of Lagrangian coherent structures (LCS)—the material boundaries in unsteady flows—can be further extracted as the ridges in stochastic FTLE fields. Similarly, uncertain flow topologies are based on the distributions of SFMs. Even more methods for uncertain flow analysis methods are likely to be developed for various applications, but the main obstacle will be the SFM computation.

SFM computation is extremely expensive and thus requires supercomputers. Currently, the only practical solution for computing SFMs is to perform Monte Carlo runs, which trace the particle stochastically in the uncertain data. However, one must trace billions or even trillions of particles for a typical analysis. For example, if the number of grid points and Monte Carlo runs is 10^6 and 10^3, respectively, and if the data has 10^3 time steps, the overall number of particles will be 10^{12}. As documented in previous studies [1], [2], it may take hours to days to run a small problem, even with GPU acceleration.

Achieving high scalability with existing parallel particle tracing algorithms in SFM computation is difficult. Two basic parallelization strategies exist: parallel-over-seeds and parallel-over-data. The parallel-over-seeds algorithms distribute particles over processes, and each process loads the required data on demand. The parallel-over-data algorithms partition the data into blocks, distribute the blocks to different processes on initialization, and exchange particles that are leaving the local blocks during the run time. However, the parallel efficiency decreases rapidly as we add more computational resources because of the flow complexity and the communication cost. In recent publications, the parallel efficiency is about 45% on 16K cores for 162M particles [3] and 35% on 16K cores for 40M particles [4]. Tracing billions or even trillions of particles at extreme scale is still challenging.

We have observed two major differences between deter-

ministic and stochastic flow map computations. First, the task dependencies in deterministic flow map computation are strict, but they can be loosened in SFM computation. By default, one must trace a particle based on its current location. In the stochastic case, the "current" location is also stochastic; thus the strong dependency can be released by transforming the problem into a probabilistic model. Second, the problem size of SFM computation is much larger. Existing parallel algorithms do not scale for the numbers of particles required by the Monte Carlo runs. The challenges are the memory footprint, the designs of task models, load balancing, and communication patterns. Solving these challenges requires a new parallel framework for SFM computation.

We propose a decoupled SFM computation that removes the time dependencies, in turn reducing communication and improving scalability. For time-varying uncertain flow data, we can compute SFMs between adjacent time steps independently on supercomputers and then compose them for any arbitrary time interval of interest. The rationale for the composition is the law of total probability. The computation is based on sparse matrix multiplication. Because the working data (two adjacent time steps) is much smaller than the whole sequence, we can duplicate the working data across processes as much as possible, so that more data are locally available. Decoupling the advection into short time intervals also shortens the travel distances of particles, and thus less communication is required. In addition, we introduce adaptive refinement over the number of Monte Carlo runs for each seed location. Experiments show that computing decoupled SFMs combined with adaptive refinement is more efficient.

In our software architecture, we adopt a novel hierarchical parallelization. On the top level, processes are subdivided into groups, each with a duplication of the working data. They are embarrassingly parallel over shuffled seed locations. Within groups, each process has a portion of data blocks, and MPI/thread hybrid parallelization is used. A dedicated thread is used to manage nonblocking interprocess communications, and a pool of threads is employed to process particle tracing tasks. Lock-free data structures are also used to manage the tasks queues. All compute cores work concurrently without any synchronization.

The task model design is also unique. The granularity of a task is a packet of particles associated with the same block. The benefits of this approach are avoiding frequent context switch in MPI/thread parallelization and enabling CPU/GPU coprocessing when GPUs are available. The philosophy of coprocessing is to schedule complex and heavy tasks for GPUs while leaving lighter tasks for CPUs. To the best of our knowledge, our system is the first such hybrid CPU/GPU implementation for parallel particle tracing problems.

We demonstrate the scalability of our system on two leadership computing facilities: Mira in Argonne National Laboratory and Titan in Oak Ridge National Laboratory. On Mira, we test the performance up to 1 million Blue Gene/Q cores over 16,384 nodes. On Titan, we test up to 131,072 AMD Opteron cores cooperating with 8,192 NVIDIA K20X GPUs. On these supercomputers, our method allows tens of billions of particles to be traced in seconds. Our system thus can help scientists analyze uncertain flows in greater detail with higher performance than previously possible. In summary, the contributions of this paper are as follows.

- A decoupled scheme that makes it possible to compute SFMs in a highly parallelized manner.
- An adaptive refinement algorithm to reduce SFM computation cost.
- A fully asynchronous parallel framework for stochastic parallel tracing based on thread pools, nonblocking communication, and lock-free data structures.
- A parallel CPU/GPU coprocessing particle tracing implementation based on the asynchronous framework.

The rest of this paper is organized as follows. We introduce the background and review related work in Section II. The decoupled and adaptive SFM computation is described in Section III, followed by the parallel framework design in Section IV. We demonstrate the application cases in Section V and then evaluate the performance in Section VI. Conclusions are drawn in Section VII.

II. Background

We formalize the concepts of stochastic flow maps and review the related work on uncertain flow visualization and parallel particle tracing.

A. Stochastic Flow Maps

We review the concepts of flow maps in deterministic data and then describe their generalization in uncertain flows.

Formally, in a deterministic flow field $\mathbf{v} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n$, the flow map ϕ maps the $(n+2)$-dimensional tuple $(\mathbf{x}_0, t_0, t_1)$ into $\mathbf{R}^n$, where n is the data dimension and t_0, t_1 are time. As illustrated in Fig. 1(a), the physical meaning of $\phi_{t_0}^{t_1}(\mathbf{x}_0)$ is the location at time t_1 of the massless particle released at the spatiotemporal location $(\mathbf{x}_0, t_0)$. Assuming $\mathbf{v}$ satisfies the Lipschitz condition, the flow map is defined by the initial value problem

$$\frac{\partial \phi_{t_0}^{t_1}(\mathbf{x}_0)}{\partial t_1} = \mathbf{v}(\phi_{t_0}^{t_1}(\mathbf{x}_0)), \text{ and } \phi_{t_0}^{t_0}(\mathbf{x}_0) = \mathbf{x}_0. \tag{1}$$

In analyses such as FTLE, the flow map is usually computed at the same resolution as that at the data discretization. Particles are seeded at every grid point (or cell center) and then traced over time until time t_1. Numerical methods, such as Euler or Runge-Kutta, are usually used in the particle tracing. Based on the definition, we can derive that

$$\phi_{t_0}^{t_2}(\mathbf{x}_0) = \phi_{t_1}^{t_2}(\phi_{t_0}^{t_1}(\mathbf{x}_0)) = \phi_{t_1}^{t_2}(\mathbf{x}_1), \tag{2}$$

where $t_0 \leq t_1 \leq t_2$.

The uncertain flow field $\mathbf{V} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n$ and its flow map Φ are stochastic. As shown in Fig. 1(b), for a given seed $(\mathbf{x}_0, t_0)$, the final location of this particle at time t_1 is a random variable denoted as $\Phi_{t_0}^{t_1}(\mathbf{x}_0)$. The probability density function of $\Phi_{t_0}^{t_1}(\mathbf{x}_0)$ is defined as

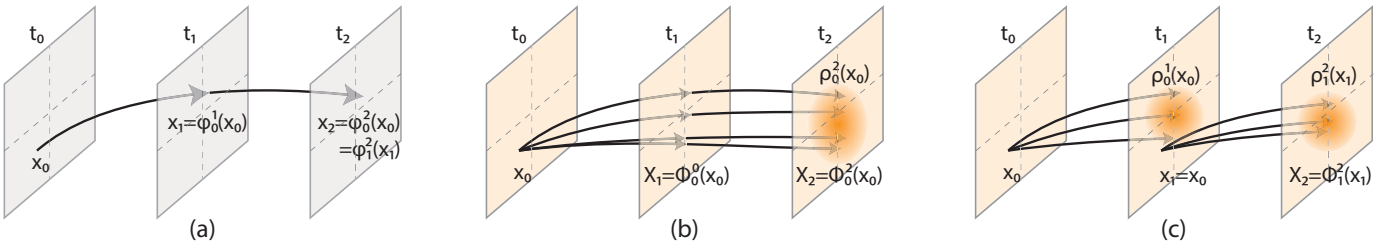

Fig. 1. (a) Flow map computation in a deterministic flow; (b) SFM computation in an uncertain flow; (c) decoupled SFM computation.

$$\rho_{t_0}^{t_1}(\mathbf{x}_0;\mathbf{x}) = Pr(\Phi_{t_0}^{t_1}(\mathbf{x}_0) = \mathbf{x}), \tag{3}$$

where ρ is a $(2n+2)$-dimensional scalar function. The usual approach is to trace a number of particles in $\mathbf{V}$ by Monte Carlo simulations and then to estimate the densities of particles. For each particle originated from $(\mathbf{x}, t)$, the final location is the solution of the stochastic differential equation

$$d\Phi_{t_0}^{t_1}(\mathbf{x}_0) = \mathbf{V}(\Phi_{t_0}^{t_1}(\mathbf{x}_0), t_1)dt_1 + \mathbf{B}(\Phi_{t_0}^{t_1}(\mathbf{x}))d\xi_{t_1}, \tag{4}$$

where $\mathbf{B}$ and ξ characterize the random disturbance. This system can be solved by Euler-Maruyama or stochastic Runge-Kutta methods. The computation of ρ is extremely expensive. Hence, we propose to alleviate the cost with a novel parallelization strategy. Similar to the property for stochastic flow maps, we have

$$Pr(\Phi_{t_0}^{t_2}(\mathbf{x}_0) = \mathbf{x_2}) = Pr(\Phi_{t_1}^{t_2}(\mathbf{x}_1) = \mathbf{x_2} \mid \Phi_{t_0}^{t_1}(\mathbf{x}_0) = \mathbf{x_1}). \tag{5}$$

We will use Eq. 5 to efficiently compute $\rho_{t_0}^{t_2}$ given that $\rho_{t_0}^{t_1}$ and $\rho_{t_1}^{t_2}$ are already known (Section III).

B. Uncertain Flow Visualization and Analysis

Uncertain flows impose a grand challenge in visualization, because they are based on two core topics—uncertainty visualization and flow visualization. Comprehensive reviews on uncertainty visualization can be found in [5], [6], and reviews on flow visualization are available in [7], [8], [9].

We categorize uncertain flow visualization techniques into two major types: Eulerian- and Lagrangian-based methods. This classification based on fluid dynamics considers flow fields at specific spatiotemporal locations and at individual moving parcels, respectively. Eulerian uncertain flow visualizations usually directly encode data into visual channel, such as colors, glyphs [10], and textures [11]. Our focus instead in this paper is on the Lagrangian-based methods that analyze transport behaviors in uncertain unsteady flows.

Lagrangian uncertain flow visualization includes topology analysis for stationary data and FTLE-based analysis for time-varying data. Otto et al. [12] extend vector field topology to 2D static uncertain flow. Monte Carlo approaches are used to trace streamlines that lead to topological segmentation. The same technique is applied to 3D uncertain flows in a later work [2]. For 3D unsteady flows, vector field topologies are no longer feasible because they are unstable and overwhelmingly complicated. FTLE and LCS are alternatives for analyzing unsteady flows. One use of FTLE in uncertain unsteady flows is FTVA [13], which is based on the variance of particles advected from the same locations over a time interval of interest. Recently, Guo et al. [1] proposed two metrics to generalize FTLE in uncertain unsteady flows: D-FTLE and FTLE-D. The former is the distribution of FTLE values that can lead to uncertain LCS extraction; the latter measures the divergence of particle distributions and has showed better results than variance-based methods. In this paper, we address the common problem of these methods: the high computational cost of Monte Carlo particle tracing.

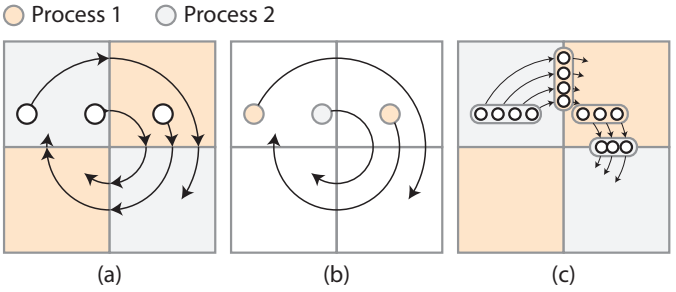

Fig. 2. Parallel particle tracing paradigms: (a) parallel-over-data; (b) parallel-over-seeds; (c) parallel-over-tasks used in this paper. The task granularity in (c) is a pack of particles associated with the same block.

C. Parallel Particle Tracing

Parallel particle tracing is a challenging problem in both the HPC and visualization communities. A comprehensive review of this topic can be found in [14]. Parallel particle tracing algorithms can be categorized into two basic types—parallel-over-data and parallel-over-seeds, as illustrated in Fig. 2. The two paradigms can also be combined for better scalability.

Parallel-over-data algorithms rely on data partitioning for load balancing. A common practice of data partitioning is to subdivide the domain into regular blocks. Peterka et al. [15] show that static round-robin block assignments with fine block partitioning can lead to good load balancing in tracing streamlines in 3D vector fields. The static load balancing can be further improved by assigning blocks based on estimated workloads [16]. Nouanesengsy et al. [3] further partition the data over time in FTLE computation. In addition to regular blocks, irregular partitioning schemes are used to improve

the load balancing. For eample, Yu et al. [17] propose a hierarchical representation of flows, which defines irregular partitions for parallel particle tracing. Similarly, mesh repartitioning algorithms are used to balance the workload across processes [18]. Our method follows the regular decomposition and round-robin assignments for intragroup parallelism.

In parallel-over-seeds algorithms, seeds are distributed over processes. Pugmire et al. [19] explore this strategy to load data blocks on demand; thus there is no communication between processes to exchange particles. Guo et al. [20] present a framework to manage the on-demand data access based on the key-value store. Fine-grained block partitioning and data prefetching are employed to improve the parallel efficiency. The parallel-over-seeds paradigm shows better performance in applications such as 3D stream surface computation [21], but it often suffers from load-balancing issues because flow behaviors are complicated and unpredictable. Work stealing has been used to improve the load balancing in 3D stream surfaces computation [22]. Mueller et al. [23] propose a work requesting approach that uses a master process to dynamically schedule the computations. In this work, we dynamically schedule the tasks between worker threads within single processes.

Hybrid methods combine both parallelization paradigms. For example, a hybrid master/worker model can be used to dynamically schedule both particles and blocks [19]. DStep [4] employs multitiered task scheduling combined with static data distribution. The framework is further extended to handle a large number of pathline tracing tasks for ensemble flow analysis [24]. In this work we must handle even larger scales of particles adaptively. Camp et al. [25] develop a hybrid implementation based on an MPI/threads programming model, which is also used in a distributed GPU-accelerated particle tracing implementation [26].

We regard our system as a hybrid method. We parallelize over data within process groups while parallelizing over seed locations across the groups. The MPI/threads model is also used with a unique task design, which is a packet of particles instead of single particles that are used by Camp et al. [25]. This model also enables us to trace massive particles on all available CPU and GPU resources simultaneously. We further compare our work with previous studies in the following sections.

Our work is also related to adaptive refinements in FTLE computation. Barakat and Tricoche [27] show that the FTLE field can be estimated by sparse samples instead of tracing densely seeded particles. An alternative approach is to sacrifice accuracy by hierarchical particle tracing [28]. These methods, however, are difficult to scale in distributed parallel environments. Our algorithm instead adapts the number of Monte Carlo runs in a full-resolution SFM computation.

III. Decoupled and Adaptive SFM Computation

In this section, we introduce the decoupled scheme and the adaptive refinement algorithm to compute SFMs. The work flows of our methods are shown in Fig. 3. We first compute decoupled SFMs and then compose them to SFMs of arbitrary time intervals. The SFMs are further used for visualization and analysis.

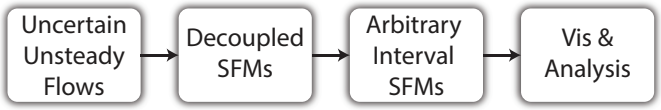

Fig. 3. The workflow of our methods.

A. Decoupled Computation of SFMs

Decoupling the particle advection of successive time steps is the key to achieving high scalability in SFM computation. Decoupling removes the time dependencies, so that we can first compute SFMs for adjacent time steps in independent runs and then compose them for arbitrary time intervals. Decoupling has two benefits. First, for adjacent time steps, the lifetimes and travel distances of particles are less than in long time periods, reducing the communication cost and the memory footprint. Second, the working datasets in independent runs are much smaller and hence easier to handle the entire data at once, leaving more memory to duplicate the working dataset across processes. Thus, data locality is improved, and less communication is required in order to exchange tasks between processes.

We illustrate the SFM decoupling in Fig. 1(c). Formally, we decouple the computation of $\rho_{t_i}^{t_j}$, given arbitrary i and j that satisfy $0 \leq i < j \leq n_t - 1$, where n_t is the number of time steps of the data. We first independently compute SFMs for adjacent time steps $\rho_{t_k}^{t_{(k+1)}}$, $0 \leq k < n_t - 2$, and then derive $\rho_{t_i}^{t_j}$ based on $\rho_{t_k}^{t_{(k+1)}}$.

The theoretic foundation of our algorithm is based on Eqs. 3 and 5. Without loss of generality, we wish to compute $\rho_{t_0}^{t_2}$ given $\rho_{t_0}^{t_1}$ and $\rho_{t_1}^{t_2}$. The scalar function $\rho_{t_0}^{t_2}$ can be written as

$$
\begin{aligned}
&\rho_{t_0}^{t_2}(\mathbf{x_0}; \mathbf{x_2}) \overset{(3)}{=} Pr(\Phi_{t_0}^{t_2}(\mathbf{x}_0) = \mathbf{x}_2) \\
&\overset{(*)}{=} \int_D \rho_{t_0}^{t_1}(\mathbf{x}_0; \mathbf{x}_1) Pr(\Phi_{t_1}^{t_2}(\mathbf{x}_1) = \mathbf{x}_2 \mid \Phi_{t_0}^{t_1}(\mathbf{x_0}) = \mathbf{x}_1) d\mathbf{x}_1 \\
&\overset{(**)}{=} \int_D \rho_{t_0}^{t_1}(\mathbf{x}_0; \mathbf{x}_1) Pr(\Phi_{t_0}^{t_1}(\mathbf{x}_0) = \mathbf{x}_1 \mid \Phi_{t_0}^{t_1}(\mathbf{x}_0) = \mathbf{x}_1) \\
&\qquad Pr(\Phi_{t_1}^{t_2}(\mathbf{x}_1) = \mathbf{x_2} \mid \Phi_{t_0}^{t_1}(\mathbf{x}_0) = \mathbf{x}_1) d\mathbf{x}_1 \\
&\overset{(5)}{=} \int_D \rho_{t_0}^{t_1}(\mathbf{x}_0; \mathbf{x}_1) \times 1 \times Pr(\Phi_{t_1}^{t_2}(\mathbf{x}_1) = \mathbf{x}_2) d\mathbf{x}_1, \\
&\overset{(3)}{=} \int_D \rho_{t_0}^{t_1}(\mathbf{x}_0; \mathbf{x}_1) \rho_{t_1}^{t_2}(\mathbf{x}_1; \mathbf{x}_2) d\mathbf{x}_1,
\end{aligned}
\tag{6}
$$

where D is the domain of the uncertain vector field $\mathbf{V}$. The rationale of (*) is the law of total probability, and (**) is based on that $\Phi_{t_0}^{t_1}(\mathbf{x}_0)$ and $\Phi_{t_1}^{t_2}(\mathbf{x}_1)$ are independent.

We discretize ρ over the same mesh as the input data mesh and store $\rho_{t_i}^{t_j}$ in the square matrix $\mathbf{P}_i^j$. The density of traced particles is estimated and stored in $\mathbf{P}$. $\mathbf{P}_i^j$ is usually sparse, and its dimension is $m \times m$, where m is the number of cells in the

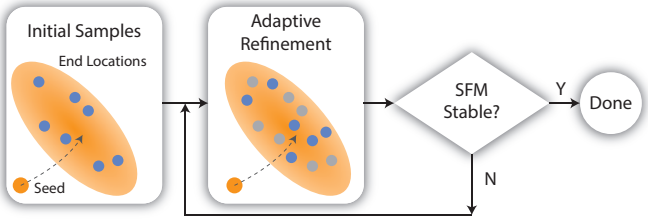

Fig. 4. Adaptive refinement.

mesh. $\mathbf{P}_i^j(p,q)$ is the transition probability of the transport from the pth to the qth cell between the ith and jth time steps. The generalized discrete form of Eq. 6 is simply matrix multiplication:

$$\mathbf{P}_i^j = \prod_{i \le k < j} \mathbf{P}_k^{k+1}. \tag{7}$$

In the matrix multiplication, we further reduce the computation and storage cost by pruning small elements in the matricies. In our implementation, we use the PETSc library [29] for sparse matrix multiplication.

B. Adaptive Refinement of SFMs

We dynamically control the number of Monte Carlo runs for each seed location in order to improve precision and reduce computational cost. Adaptive refinement is based on the observation that transport behaviors in flows are usually coherent. Barakat and Tricoche [27] propose an adaptive refinement of deterministic flow maps based on reconstruction of sparse samples. Denser seeds are needed in regions with rich flow features, and fewer samples are necessary in less complicated parts. However, the technique is hard to scale in parallel. We instead use densely seeded particles and adaptively control the number of stochastic runs for each seed.

The adaptive refinement for each seed is illustrated in Fig. 4. In the kth iteration, a batch of particles is traced from the seed $\mathbf{x}_0$, and the density of these particles is estimated as $D_k(\mathbf{x}_0;\mathbf{x})$. The loop exits if $D_{k-1}(\mathbf{x}_0;\mathbf{x})$ and $D_k(\mathbf{x}_0;\mathbf{x})$ are statistically identical, otherwise enters the next iteration. Our criterion to stop iteration is the difference of information entropies between $D_{k-1}(\mathbf{x}_0;\mathbf{x})$ and $D_k(\mathbf{x}_0;\mathbf{x})$. The information entropy of a random variable X is defined as

$$H(X) = -\sum_{l=0}^{m} P(x_l)\log(P(x_l)), \tag{8}$$

where m is the number of probabilistic states (number of cells in this case) and $P(x_l)$ is the probability of state x_l. We then evaluate $H(D_{k-1}(\mathbf{x}_0;\mathbf{x}))$ and $H(D_k(\mathbf{x}_0;\mathbf{x}))$. If $|H(D_k) - H(D_{k-1})|$ is greater than a preset threshold, we add more samples; otherwise we stop the iteration and store $D_k(\mathbf{x}_0;\mathbf{x})$ in the sparse matrix. Fig. 5(a-d) show a comparison of adaptive refinement with fixed numbers of Monte Carlo runs in the uncertain Isabel dataset. More particles are traced in the hurricane eye regions, while fewer are sampled in other regions. Comparing (b) with (c) and (d), we can see that the entropy field generated by adaptive refinement is more similar to the results generated with large numbers of samples. Based on this result, we conclude that adaptive refinement can achieve better precision with fewer of particles.

IV. Software Architecture Design

Our parallel particle tracing framework exploits hierarchical parallelization. At the top level, upon initialization, the processes are divided into groups. In the intergroup level, we parallelize over seed locations. Each group duplicates the working data and traces a different set of seeds. Inside each group, we parallelize over data. Each process has a portion of data blocks. A novel task model based on MPI/thread hybrid parallelization is used. The rationale for our hierarchy is based on decoupled and adaptive SFM computation. First, the decoupling makes it possible to have higher degrees of data duplication for better scalability because the working data of two adjacent time steps are smaller than that of the whole dataset. Second, the adaptive refinement allows asynchronous processing, which also boosts the scalability of parallel particle tracing.

We further implement a novel design of task model—packets of particles—to achieve high parallel efficiency in the MPI/thread model. Within each process, the tasks are scheduled and processed by a pool of threads in parallel. The interprocess task exchange is managed by a dedicated thread, which handles nonblocking MPI communication. Lock-free data structures are used to exchange data between threads. In general, this design is fully asynchronous—communication and computation are overlapped, and threads are synchronization-free. This design improves data locality and enables CPU/GPU coprocessing.

A. Initialization

Because the decoupled SFM computation yields smaller working data, typically two adjacent time steps, it is possible to duplicate data in order to improve data locality. We first partition the data into blocks and then determine how many processes to assign to each group for the given memory limit. For example, given 4 processes, the total number of blocks is 64, and the maximum number of blocks per process is 32. Then we create 2 process groups.

Within groups, the blocks are distributed across processes. As in Peterka et al. [15], we statically assign blocks to processes by a round-robin scheme. Each process is in charge of one or more blocks. In addition, threads and lock-free data structures for task exchanging are also created upon the initialization.

B. Task Model

Fig. 6 illustrates the task model. We define a task as a tuple (blkID, type, particles[]), where particles[] is a packet of particles associated with only on block (blkID). Notice that each task is associated with one block (blkID). The granularity of a task is one or more particles, up to a given limit. Each

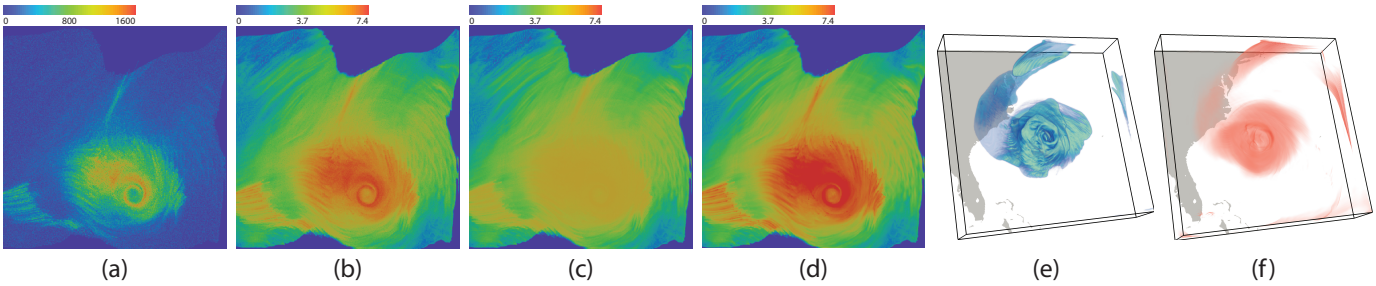

Fig. 5. Experiment results of the uncertain Isabel data: (a) the number of Monte Carlo runs with adaptive refinement; (b) the entropy of SFMs computed with adaptive refinement; (c) and (d) the entropy of SFMs computed with 256 and 2,048 runs, respectively; (e) uncertain LCSs; (f) the FTLE-D. (a)-(d) are slice rendering and (e)-(f) are volume renderings.

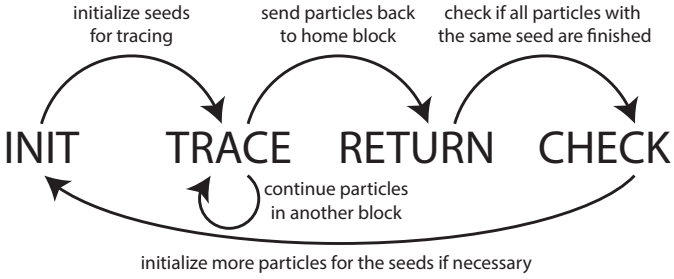

Fig. 6. Task model.

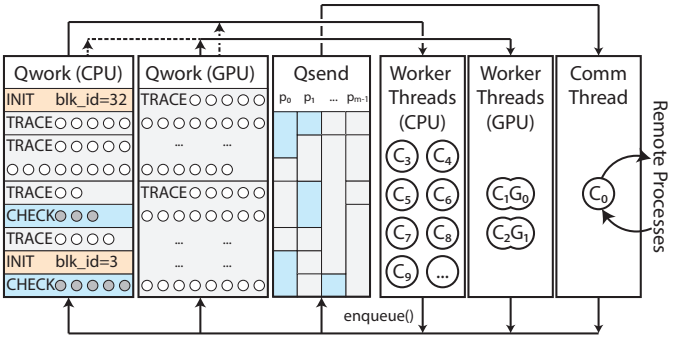

Fig. 7. Thread model.

particle is a tuple $(\mathbf{x}_0, \mathbf{x})$ consisting of its initial and current spatiotemporal locations, respectively.

Four types of tasks are depicted in the model: initialization (`INIT`), tracing (`TRACE`), return (`RETURN`), and checking (`CHECK`).

- `INIT` tasks are used to initialize particles for a list of seed locations in the given block. Particles are created either by the system for bootstrapping or by the `CHECK` tasks when more particles are necessary to refine the SFMs.
- `TRACE` tasks start or continue to trace a pack of particles that are not finished yet. If particles are moving out of the current block, new `TRACE` tasks associated with the target blocks are created.
- `RETURN` tasks are created by `TRACE` tasks to send finished particles to their home blocks.
- `CHECK` task checks termination for particles released at the same location $\mathbf{x}_0$. The density is written to the output sparse matrix if it is converged; otherwise new `INIT` tasks are created to refine the density.

C. Thread Model

We use multithreading for parallelism within a single process. Fig. 7 illustrates the thread model in our design. Two types of threads exist: the communication threads and the worker threads. Because each process is assigned only a subset of blocks, the worker threads can handle only messages that are associated with the blocks they have. Several lock-free producer-consumer queues are used to schedule and exchange tasks between threads. There are two groups of queues: the work queues and the send queues that keep the pending tasks for the local and remote processes, respectively.

Algorithm 1 Worker thread loop. The process_task() function processes a input task and returns a list of new tasks to continue the computation.

```
while !all_done do
    if Q_work.pop(task) then
        new_tasks[] = process_task(task)
        for all task in new_tasks[] do
            comm.enqueue(task.blkID, task)
```

Algorithm 1 shows the pseudo code of the worker thread main loop. The worker threads function as both producers and consumers. Worker threads consume tasks and also produce new tasks to deliver particles to their next or final destinations. The new task is enqueued to the work queue if the current process owns the destination block; otherwise the task is appended to the send queues. The `enqueue()` function (Algorithm 2) simplifies the task routing.

The maximum number of particles for each task (`max_size_CPU`), which defines the granularity of a `TRACE` task, is the most important parameter that determines the scalability of the thread pool. Larger task granularity leads to load imbalance because there are fewer concurrent tasks and some threads are starving. On the contrary, a smaller task size can result in more context switches and more contention for the task queues. Fig. 8(a) shows a scalability benchmark using different `max_size_CPU` values. In this experiment, 64 is

Algorithm 2 Enqueue task

```
function ENQUEUE(blkID, task)
    i ←blkID_to_rank(blkID)
    if i=comm.rank then
        if task.size≥max_size_GPU then
            split_tasks[]=task.split(max_size_GPU)
            enqueue_all(split_tasks[])
        else if task.size≥min_size_GPU then
            Q_work^(GPU).push(task)
        else if task.size≥max_size_CPU then
            split_tasks[]=task.split(max_size_CPU)
            enqueue_all(split_tasks[])
        else
            Q_work^(CPU).push(task)
    else
        Q_send^i.push(task)
```

Algorithm 3 Communication thread loop

```
while !all_done do
    for all i in comm.world do                ▷ outgoing tasks
        if Q_send^i.pop_bulk(tasks, max_size_send) then
            comm.isend(i, serialize(tasks))
    while comm.iprobe() do                    ▷ incoming tasks
        tasks = unserialize(comm.recv())
        for all task in tasks do
            enqueue(task.blkID, task)
    comm.iexchange(all_done)                  ▷ exchange status
```

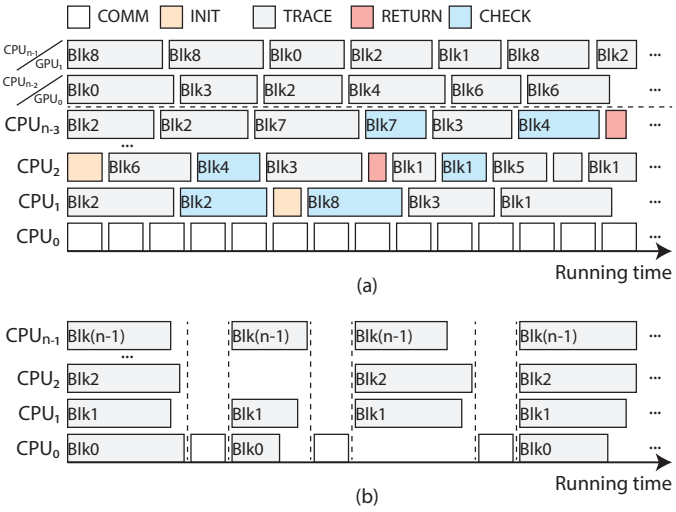

Fig. 9. Gantt chart of (a) our task model and (b) the bulk synchronous parallel model. Each row represents a thread.

the optimal selection. A similar parameter (`max_size_GPU`) needs to be configured when a GPU is available for coprocessing with CPUs. The principle to set `max_size_GPU` is to have approximately equivalent processing time on GPUs as that on CPUs, so `max_size_GPU` is usually larger than `max_size_CPU`. More details on the parameter setting in CPU/GPU coprocessing are in Section IV-E.

The communication thread consumes tasks in the send queues by sending them to the destination process and enqueues tasks that are received from remote processes to work queues. Our thread model uses a dedicated thread for communication, a common practice in MPI/thread hybrid parallelization, such as that in Charm++ [31]. The communication thread in Charm++, however, handles communication on behalf of worker threads, but the worker threads in our design do not involve any communication operations. In addition, our communication thread schedules tasks for load balancing and CPU/GPU coprocessing.

D. Two-Tiered Asynchronous Communication

We adopt a two-tiered asynchronous design. First, the interprocess communication overlaps the computation by using a dedicated communication thread. Second, the communication thread uses MPI nonblocking communications to further reduce the delays. Specifically, tasks are exchanged between blocks across processes by the two-tiered asynchronous communication in to overlap the computation. Each process has a dedicated communication thread to send and receive messages from remote processes. The communication thread executes and manages nonblocking MPI requests without any waits.

The pseudo code of the communication thread main loop is listed in Algorithm 3. Each process maintains a list of lock-free send queues $\{Q^i_{\text{send}}\}_m$, where i is the destination rank and m is the number of processes. The tasks in the send queues are pushed by the worker threads via `enqueue()` calls. In every iteration of the loop, the communication thread tries to dequeue a bulk of tasks from each Q^i_{send}. The list of tasks is then serialized and sent to the destination process by nonblocking send (`MPI_Isend`). The incoming messages are received by `MPI_Recv` if they are probed by `MPI_Iprobe`. The loop exits when all tasks across all processes are finished. A nonblocking version of Francez's algorithm [32] is implemented for distributed termination, as in a previous parallel particle tracing study [22].

We use $m-1$ send queues for better performance. In our design, a set of tasks with the same destination rank is obtained with the `pop_bulk()` function in the lock-free queue. Thus, we can send a larger message that contains multiple tasks, instead of multiple smaller messages each contains one single task. We do so because larger message size usually yields better performance.

The two-tiered asynchronous design enables the full overlap between computation and communication. As illustrated in Fig. 9(a), the communication thread (CPU_0) and the worker threads (other CPUs) and the communication thread work concurrently without any explicit synchronization. In Section VI-C we compare our design with communication models in previous parallel particle tracing studies.

E. CPU/GPU Coprocessing

The thread pool model enables hybrid CPU/GPU parallelization, which fully utilizes the computation power of both

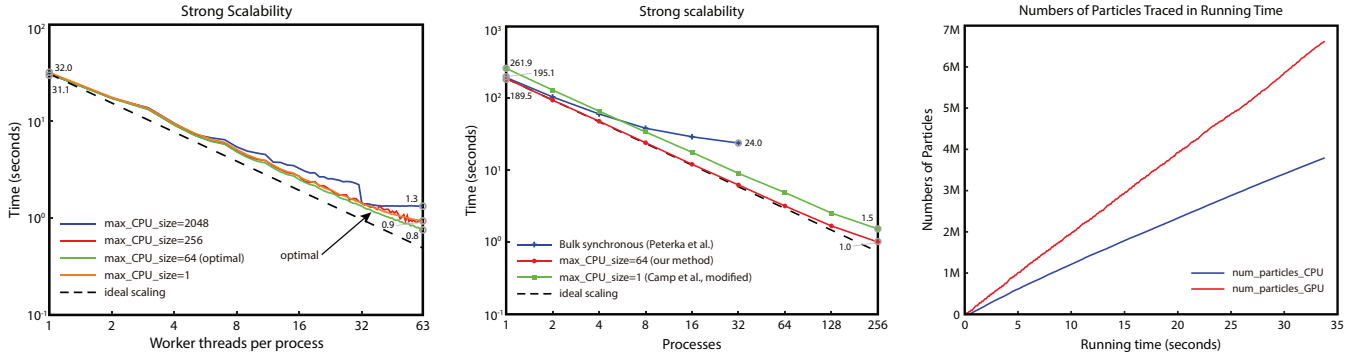

Fig. 8. (a) Benchmark of the flow map computation in tornado simulation data (32K particles) with different `max_size_CPU` and different numbers of worker threads per process on 32 Blue Gene/Q nodes. A proper selection of `max_size_CPU` leads to better performance and scalability. (b) Performance comparison with two existing parallel particle tracing methods ([30] and [25]). (c) Number of particles traced in running time with GPU/GPU coprocessing.

CPUs and GPUs in compute nodes. We design strategies to schedule tasks on CPUs and GPUs.

Our task-scheduling strategy is to fill GPUs with larger tasks and assign complex or small tasks for CPUs. GPUs can be seen as SIMD processors, which are suitable for handling a batch of tasks simultaneously. However, the data movement cost between the CPU and GPU is significant. Specifically, the particles must be transferred to the GPU memory before they are traced, and they have to be copied back to the main memory for further processing. The clock speed of GPUs is also slower than that of CPUs. Thus, the overall performance drops if there are too few particles for a batch. This phenomenon was observed by Camp et al. [26] in distributed and parallel environments.

We associate a GPU worker thread (running on the CPU) with each GPU. The data blocks in the main memory are copied into GPU in the initialization stage. A designated GPU task queue is also set up for task scheduling. In the `enqueue()` function, larger tasks and smaller tasks are pushed into the GPU and CPU queues, respectively.

Similar to the rationale of `max_size_CPU` for CPU workers, we also need to limit the task size for the GPU, that is, `min_size_GPU`. In principle, the running time cannot be too long, in order to keep load balanced. We usually set `max_size_CPU` $\leq$ `min_size_GPU` $<$ `max_size_GPU`.

Although we have two different work queues for CPUs and GPUs, the tasks do not have to be processed by their designated processors. A CPU worker thread can dequeue a task from the GPU queue when the thread is starving, and vice versa for GPU worker threads. When a task T in the GPU queue is processed by a CPU worker thread, T may be split before further processing. Because the task size is usually greater than `max_size_CPU`, the incoming task is cut into two subtasks T_1 and T_2. Task T_1 has size `max_size_CPU`, and task T_2 is the rest. T_1 is processed with the current CPU worker thread, and T_2 is enqueued to worker queues for further processing. Notice that T_2 may or may not qualify as a GPU task depending on its size.

When a task T in the CPU queue is obtained by a GPU worker thread, it may or may not be processed with the associated GPU. First, if the size of the `TRACE` task T is smaller than `min_size_GPU`, it is still handled by a CPU. Second, the `INIT` or `RETURN` are also processed on a CPU.

F. Implementation

We implemented the prototype system with C++11. MPI is used for interprocess communication. For each process, the worker threads are created with Pthreads, and the parent thread plays the role of the communication thread. We use a lock-free concurrent queue implementation [33] to exchange tasks between threads. Because only one thread makes MPI calls, we use the `MPI_THREAD_FUNNELED` mode on initialization. DIY [30] is used for domain decomposition. The Block I/O Layer (BIL) library [34] is used to efficiently load disjoint block data across different files and processes collectively. We also implemented a thread-specific random number generator for stochastic particle tracing, because the random number generator in the C++ standard library does not scale multiple threads in our experiments. The GPU code is written in CUDA. Upon initialization, the data blocks are copied to the GPU memory, and then a buffer that can fit `max_size_GPU` particles is created. Particles in the `TRACE` task are copied to the GPU and then copied back after they are traced. After the computation, we store the SFMs in a sparse matrix that is managed by the PETSc library [29].

V. Application Results

We applied our method to two weather simulation datasets with uncertainties: uncertain Hurricane Isabel data and ensemble Weather Research and Forecasting (WRF) data.

A. Input Data

Uncertainty arises in the Hurricane Isabel data from temporal down-sampling. In climate and weather simulations, a common practice is to dump average data hourly or daily instead of every time step. Such data down-sampling reduces the I/O cost but sacrifices accuracy. We follow Chen et al. [35] who use quadratic Bezier curves to quantify the uncertainty of the original Hurricane Isabel data which is from the IEEE

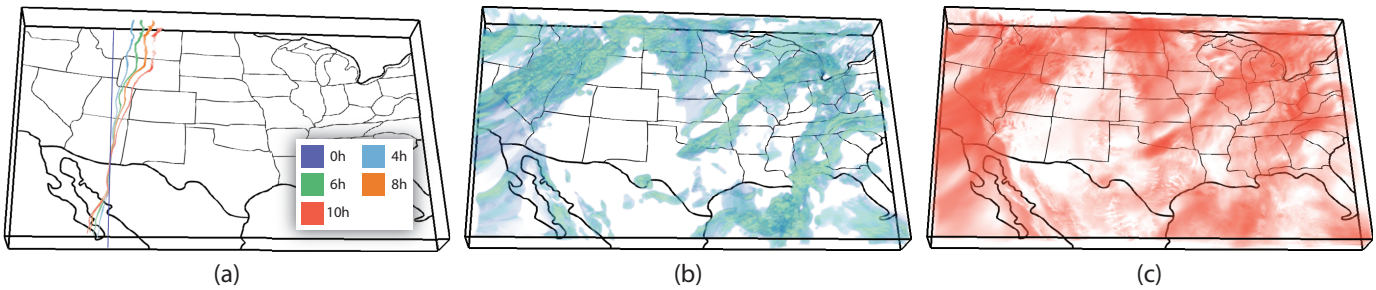

Fig. 10. Experiment results of the WRF ensemble simulation data: (a) uncertain source-destination queries; (b) uncertain LCSs; (c) FTLE-D.

Visualization Contest 2004. The spatial resolution of the original data is $500 \times 500 \times 100$. The down-sampled dataset we use in the experiment keeps the full spatial resolution but aggregates every 12 time steps into one. The parameters of the quadratic Bezier curves and the Gaussian error are used to reconstruct the uncertain flow field for SFM computation.

The uncertainty of the ensemble WRF data arises from averaging the ensemble members. The input data, courtesy of the National Weather Service, is simulated with the High Resolution Rapid Refresh (HRRR) model [36]. The model is based on the WRF model and assimilates observations from National Oceanic and Atmospheric Administration (NOAA) and other sources. The spatial resolution of the model is $1799 \times 1059 \times 40$, and we use 10 ensemble members with 15 hourly averaged outputs for our experiment. The uncertainty is modeled as Gaussian—the mean and covarances of the ensemble members are computed for every grid point location.

B. Uncertain Source-Destination Queries

Fig. 10(a) shows the uncertain source-destination query results. We create particles along a line in the domain and visualize the distributions of these particles after every hour by volume rendering. The visualization results show that the uncertainties of SFMs grow as the advection time increases. We can also see that the uncertainty of transport behavior in the mountain areas is greater than in the plains. This phenomenon matches the fact that numerical weather forecasts are more unstable in mountains.

C. Uncertain FTLE and LCS Visualization

FTLE and LCS are the most important tools for analyzing deterministic unsteady flow. The FTLE was proposed by Haller [37], and it measures the convergence or divergence for the time interval of interest. Recently, Guo et al. [1] generalized FTLE and LCS to analyze uncertain unsteady flows based on SFMs. Three new concepts were introduced: D-FTLE (distributions of FTLE), FTLE-D (FTLE of distributions), and U-LCS (uncertain LCS). We compute FTLE-D and U-LCS from the uncertain Isabel data and the WRF ensembles in Fig. 5 and Fig. 10, respectively.

The FTLE-D and U-LCS in Fig. 5(e) and (f) show connective bands of the uncertain Isabel data. The spiral arm that extends to the east coast separates two different motions: the flow going upwards and the flow keeping their original levels. Because there is more uncertainty in updraft and downdraft flows, the boundary of the two features is fuzzy, as shown in the U-LCS and the FTLE-D.

In the WRF ensembles, we can also observe that the upward and downward air flows lead to uncertainties in U-LCS and FTLE-D. These are due mainly to the land surface variability. We can see four distinct regions in Fig. 10(b) and (c): the on-shore flow from the Pacific Ocean to the Cascade mountains, a cold front from Oklahoma to Dakotas, and two unstable troughs in the Midwest and the East. The visualization results of FTLE-D and U-LCS, which are confirmed by meteorologists, highlight these unstable zones.

VI. Performance Evaluation

We study the scalability of our methods on two supercomputers: Mira and Titan. We also compare our parallel particle tracing scheme with previous studies.

A. Scalability Study on the Blue Gene/Q Systems

We conducted scalability study on Mira, an IBM Blue Gene/Q system at Argonne National Laboratory. The theoretical peak performance of Mira is 10 petaflops. Each compute node has 16 1.6 GHz PowerPC A2 cores, which support 64 hardware threads in total. The memory on each node is 16 GB, and the interconnect is a proprietary 5D torus network.

To maximize the utilization of computation nodes, we run one MPI process on each node, with one communication thread and 63 worker threads for computation. These choices are based on the experiments in Section IV-C. We limit the memory for data blocks to 1 GB per process, so we have 4 and 16 processes per group for the uncertain Isabel data and the ensemble WRF data, respectively. For the uncertain Isabel data, we use both fixed numbers of Monte Carlo runs and adaptive refinements for comparison. For the fixed sampling, the number of runs is 256; thus, the total number of particles is about 6.5 billion.

Fig. 11(a) and (c) show the timings of SFM computation on both datasets with different numbers of processes on Mira. Ideal scaling curves based on linear speedup are shown for reference. From the benchmark we can see that the speedup is nearly linear. The parallel efficiency of 4K, 8K, and 16K processes are 92%, 85%, and 72%, respectively. The main reason for this scalability is the decoupled SFM computation that removes the time dependency. Because we need to load

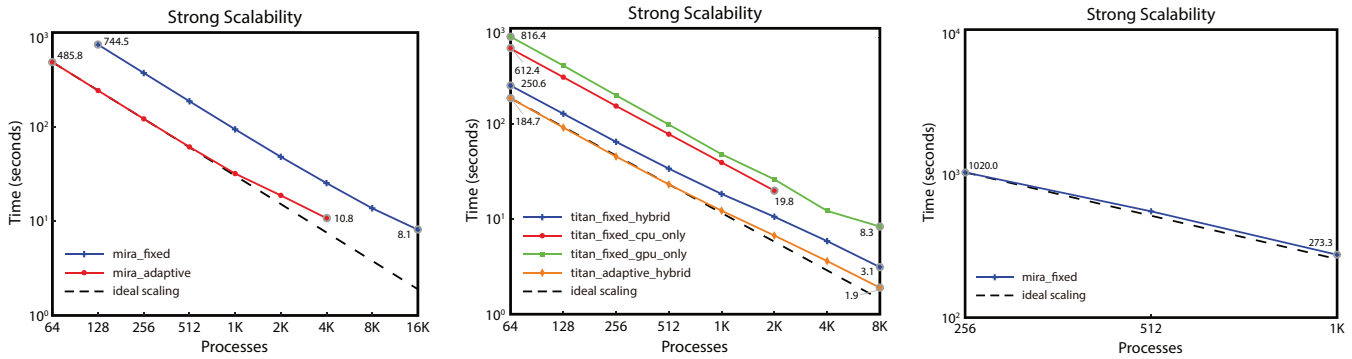

Fig. 11. Strong scalability studies of our method: (a) uncertain Isabel data on Mira; (b) uncertain Isabel data on Titan; (c) ensemble WRF data on Mira.

only two adjacent time steps at once, we can duplicate more working data for less communication. Fig. 11(a) also shows that adaptive refinement reduces the computation time, compared with fixed sampling.

B. Scalability Study on CPU/GPU Hybrid Architectures

We benchmarked the CPU/GPU coprocessing on Titan, which is a Cray XK7 supercomputer at Oak Ridge National Laboratory. Titan has 18,688 compute nodes, each equipped with an AMD Opteron 6274 16-core CPU that operates at 2.2 GHz with 32 GB of main memory. In addition to the CPU, each node also contains an NVIDIA Tesla K20X GPU with 6 GB memory. The number of CUDA cores on a single GPU is 2,688, running at 732 MHz.

Fig. 11(b) shows the strong scalability benchmark on the uncertain Isabel dataset. The problem size is the same as that on Mira, namely 6.5 billion particles. In the experiments, we fully used the CPU resources by running 15 worker threads and one communication thread per process on each node. In the CPU/GPU coprocessing mode, one of the worker threads managed the GPU. We conducted three runs to study the effectiveness of CPU/GPU hybrid parallelization: the pure CPU mode, the pure GPU mode, and the hybrid mode. The pure GPU mode is used for comparison only. In this mode, only one worker thread is used, and all tasks are conducted on the GPU regardless of task size; that is, `min_GPU_size` is zero.

Results show that the computation time of the hybrid mode is about $2.5\times$ faster than the pure CPU mode. We refer to Camp et al. [26] who report a speedup of $1\times$ to $10.5\times$ on a distributed-memory GPU particle tracer compared with a CPU-only code on 8 nodes. Based on the previous studies, we believe that our $2.5\times$ speedup is promising. Our hybrid parallelization design enables the full use of available hardware resources on compute nodes, including all CPU and GPU cores. The scheduling of CPUs and GPUs is also adaptive, capable of balancing working time between CPU and GPU workers. Moreover, the hybrid implementation is scalable up to 131,072 Opteron cores with 8,192 NVidia K20 GPUs in our test. At this scale, tracing billions of particles only spends less than 10 seconds.

C. Comparison with Existing Parallel Particle Tracing Algorithms

We compared our method with existing parallel particle tracing algorithms. The baseline approaches are these of Peterka et al. [30] and Camp et al. [25]. Both algorithms partition data into blocks for parallel processing and use MPI/thread hybrid parallelization. We implemented these algorithms and compared their performance on the same dataset and problem size. In the experiment, we used the deterministic tornado dataset and 32 threads per process for computation. Only one process group was used, so there is no data duplication for the comparison. The timings with respect to different numbers processes are shown in Fig. 8(b), and we can see that our method outperforms the others.

The parallel model used by Peterka et al. [30] is bulk synchronous (Fig. 9(b)). In this model, each block of data is associated with a thread in one single process. The particles are traced in the current block until they cross the block bounds, and then they are exchanged between neighbor blocks collectively. Compared with the bulk synchronous parallel model, our model does not associate blocks with threads. We also fully overlap the communication and computation in our framework.

The thread pool pattern is used in Camp et al. [25], but the major difference in our design is the task model and the software design. In Section IV-C, we showed that our task model yields fewer context switches and enables the CPU/GPU coprocessing. In addition, we use lock-free data structures and two-tiered asynchronous communication for intra and interprocess task exchange.

VII. Conclusions and Future Work

In this paper, we presented a scalable SFM computation method for uncertain flow visualization and analysis. The keys to achieving high scalability are the decoupled and adaptive algorithms, the MPI/thread hybrid parallelization, and the unique task design that assembles packets of particles. The decoupling allows us to compute SFMs of adjacent time steps and then compose them together. The number of stochastic runs can be adaptively configured for better efficiency and precision. We

parallelize over tasks, which are packets of particles, to achieve high efficiencies in the MPI/thread hybrid programming. Our parallelization design also enables CPU/GPU coprocessing when GPUs are available. Results show that our method can help scientists analyze uncertain flows in greater detail with higher performance than previously possible.

We would like to extend our work to support more many-core architectures, such as the Intel Xeon Phi. The data localities can be also improved in NUMA architectures. We would also like to incorporate more uncertain flow analysis tools, such as uncertain topology analysis. Our algorithms could also be used in in situ flow analysis frameworks in the future.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. This work is also supported by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

REFERENCES

[1] H. Guo, W. He, T. Peterka, H.-W. Shen, S. M. Collis, and J. J. Helmus, "Finite-time Lyapunov exponents and Lagrangian coherent structures in uncertain unsteady flows," *IEEE Trans. Vis. Comput. Graph.*, vol. 22, 2016, to appear.

[2] M. Otto, T. Germer, and H. Theisel, "Uncertain topology of 3D vector fields," in *Proceedings of IEEE Pacific Visualization Symposium 2011*, 2011, pp. 67–74.

[3] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka, "Parallel particle advection and FTLE computation for time-varying flow fields," in *SC12: Proc. ACM/IEEE Conference on Supercomputing*, 2012, pp. 61:1–61:11.

[4] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson, "Simplified parallel domain traversal," in *SC11: Proceedings of the ACM/IEEE Conference on Supercomputing*, 2011, pp. 10:1–10:11.

[5] C. R. Johnson and A. R. Sanderson, "A next step: Visualizing errors and uncertainty," *IEEE Comput. Graph. Appl.*, vol. 23, no. 5, pp. 6–10, 2003.

[6] K. Brodlie, R. AllendesOsorio, and A. Lopes, "A review of uncertainty in data visualization," in *Expanding the Frontiers of Visual Analytics and Visualization*, J. Dill, R. Earnshaw, D. Kasik, J. Vince, and P. C. Wong, Eds. Springer London, 2012, pp. 81–109.

[7] R. S. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. H. Post, and D. Weiskopf, "The state of the art in flow visualization: Dense and texture-based techniques," *Comput. Graph. Forum*, vol. 23, no. 2, pp. 203–222, 2004.

[8] F. H. Post, B. Vrolijk, H. Hauser, R. S. Laramee, and H. Doleisch, "The state of the art in flow visualization: Feature extraction and tracking," *Comput. Graph. Forum*, vol. 22, no. 4, pp. 1–17, 2003.

[9] A. Pobitzer, R. Peikert, R. Fuchs, B. Schindler, A. Kuhn, H. Theisel, K. Matkovic, and H. Hauser, "The state of the art in topology-based visualization of unsteady flow," *Comput. Graph. Forum*, vol. 30, no. 6, pp. 1789–1811, 2011.

[10] C. M. Wittenbrink, A. Pang, and S. K. Lodha, "Glyphs for visualizing uncertainty in vector fields," *IEEE Trans. Vis. Comput. Graph.*, vol. 2, no. 3, pp. 266–279, 1996.

[11] R. P. Botchen, D. Weiskopf, and T. Ertl, "Texture-based visualization of uncertainty in flow fields," in *Proceedings of IEEE Visualization 2005*, 2005, pp. 647–654.

[12] M. Otto, T. Germer, H.-C. Hege, and H. Theisel, "Uncertain 2D vector field topology," *Comput. Graph. Forum*, vol. 29, no. 2, pp. 347–356, 2010.

[13] D. Schneider, J. Fuhrmann, W. Reich, and G. Scheuermann, "A variance based FTLE-like method for unsteady uncertain vector fields," in *Topological Methods in Data Analysis and Visualization II*, ser. Mathematics and Visualization, R. Peikert, H. Hauser, H. Carr, and R. Fuchs, Eds. Springer, 2011, pp. 255–268.

[14] E. W. Bethel, H. Childs, and C. Hansen, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, 2012.

[15] T. Peterka, R. B. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang, "A study of parallel particle tracing for steady-state and time-varying flow fields," in *IPDPS11: Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2011, pp. 580–591.

[16] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen, "Load-balanced parallel streamline generation on large scale vector fields," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 1785–1794, 2011.

[17] H. Yu, C. Wang, and K.-L. Ma, "Parallel hierarchical visualization of large time-varying 3D vector fields," in *SC07: Proceedings of the ACM/IEEE Conference on Supercomputing*, 2007, pp. 24:1–24:12.

[18] L. Chen and I. Fujishiro, "Optimizing parallel performance of streamline visualization for large distributed flow datasets," in *Proceedings of IEEE Pacific Visualization Symposium 2008*, 2008, pp. 87–94.

[19] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber, "Scalable computation of streamlines on very large datasets," in *SC09: Proceedings of the ACM/IEEE Conference on Supercomputing*, 2009, pp. 16:1–16:12.

[20] H. Guo, J. Zhang, R. Liu, L. Liu, X. Yuan, J. Huang, X. Meng, and J. Pan, "Advection-based sparse data management for visualizing unsteady flow," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2555–2564, 2014.

[21] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy, "Parallel stream surface computation for large data sets," in *LDAV12: Proceedings IEEE Symposium on Large Data Analysis and Visualization*, 2012, pp. 39–47.

[22] K. Lu, H. Shen, and T. Peterka, "Scalable computation of stream surfaces on large scale vector fields," in *SC14: Proceedings of the ACM/IEEE Conference on Supercomputing*, 2014, pp. 1008–1019.

[23] C. Mueller, D. Camp, B. Hentschel, and C. Garth, "Distributed parallel particle advection using work requesting," in *LDAV13: Proceedings IEEE Symposium on Large Data Analysis and Visualization*, 2013, pp. 109–112.

[24] H. Guo, X. Yuan, J. Huang, and X. Zhu, "Coupled ensemble flow line advection and analysis," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 12, pp. 2733–2742, 2013.

[25] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy, "Streamline integration using MPI-hybrid parallelism on a large multicore architecture," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 11, pp. 1702–1713, 2011.

[26] D. Camp, H. Krishnan, D. Pugmire, C. Garth, I. Johnson, E. W. Bethel, K. I. Joy, and H. Childs, "GPU acceleration of particle advection workloads in a parallel, distributed memory setting," in *EGPGV13: Proceedings of Eurographics Parallel Graphics and Visualization Symposium*, 2013, pp. 1–8.

[27] S. S. Barakat and X. Tricoche, "Adaptive refinement of the flow map using sparse samples," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 12, pp. 2753–2762, 2013.

[28] M. Hlawatsch, F. Sadlo, and D. Weiskopf, "Hierarchical line integration," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 8, pp. 1148–1163, 2011.

[29] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[30] T. Peterka, R. B. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri, "Scalable parallel building blocks for custom data analysis," in *LDAV11: Proceedings IEEE Symposium on Large Data Analysis and Visualization*, 2011, pp. 105–112.

[31] L. V. Kale and S. Krishnan, "Charm++: Parallel programming with message-driven objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[32] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 42–55, 1980.

[33] C. Desrochers, "A fast multi-producer, multi-consumer lock-free concurrent queue for C++11," https://github.com/cameron314/concurrentqueue.

[34] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. B. Ross, "Toward a general I/O layer for parallel-visualization applications," *IEEE Computer Graphics and Applications*, vol. 31, no. 6, pp. 6–10, 2011.

[35] C.-M. Chen, A. Biswas, and H.-W. Shen, "Uncertainty modeling and error reduction for pathline computation in time-varying flow fields," in *Proceedings of IEEE Pacific Visualization Symposium 2015*, 2015, pp. 215–222.

[36] C. Alexander, S. S. Weygandt, D. C. D. S. Benjamin, T. G. Smirnova, E. P. James, M. H. P. Hofmann, J. Olson, and J. M. Brown, "The high-resolution rapid refresh: Recent model and data assimilation development towards an operational implementation in 2014," in *Proceedings of 26th Conference on Weather Analysis and Forecasting / 22nd Conference on Numerical Weather Prediction*. American Meterological Society, 2014.

[37] G. Haller, "Distinguished material surfaces and coherent structures in three-dimensional fluid flows," *Physica D: Nonlinear Phenomena*, vol. 149, no. 4, pp. 248–277, 2001.