Parallel Particle Advection Bake-Off for Scientific Visualization Workloads

Roba Binyahib
University of Oregon
roba@cs.uoregon.edu

David Pugmire
Oak Ridge National Laboratory
pugmire@ornl.gov

Abhishek Yenpure
University of Oregon
abhishek@uoregon.edu

Hank Childs
University of Oregon
hank@uoregon.edu

***Abstract*—There are multiple algorithms for parallelizing particle advection for scientific visualization workloads. While many previous studies have contributed understanding about an individual algorithm, our study aims to provide a holistic understanding of how algorithms perform relative to each other on various workloads. To accomplish this, we consider four popular parallelization algorithms and run a "bake-off" study (i.e., an empirical study) to identify the best matches for each. The study includes 216 tests, going to a concurrency of up to 8192 cores and considering data sets as large as 34 billion cells with 300 million particles. Overall, our study informs three important research questions: (1) which parallelization algorithms perform best for a given workload?, (2) why?, and (3) what are the unsolved problems in parallel particle advection? In terms of findings, we find that seeding box is the most important factor in choosing the best algorithm, and also that there is significant opportunity for improvement in execution time, scalability, and efficiency.**

***Index Terms*—Scientific visualization, particle advection, flow visualization, parallel processing**

I. Introduction

Within the field of scientific visualization, the techniques dedicated to understanding vector field behavior are called "flow visualization" techniques. The types of flow visualization techniques are varied. One well-known form involves seeding particles in a volume and animating the paths of these particles as they flow through the volume. A related approach ("streamlines") avoids animation, and instead plots the entire path that each particle follows all at one time. Other flow visualization techniques use particle trajectories as building blocks. One example is stream surfaces, which starts with a curve and considers the surface traced out as the curve is displaced by the flow. In practice, particles are seeded along the curve and the surface is made up of their trajectories. Another example is Finite-Time Lyanupov Exponents (FTLE), which considers a volume as a series of neighborhoods and assigns a scalar value to each neighborhood. For each neighborhood, the scalar value is determined by placing several particles within the neighborhood, and then measuring how much these particles separate. Overall, there is a rich space of flow visualization techniques, these techniques utilize particle trajectories, and their workloads of which particle trajectories to calculate are very diverse.

Particle trajectories are typically calculated using particle advection, which displaces a massless particle based on a vector field. A trajectory is calculated iteratively, i.e., in a series of steps. After a particle is placed at some seed position, X_0, the first step displaces the particle to some new position, X_1. The second step displaces the particle from X_1 to X_2, and so on. Each displacement involves solving an ordinary differential equation, typically (in scientific visualization) with a Runge-Kutta [1] operation. The process can terminate for a variety of reasons, including traveling a fixed distance, a fixed amount of time having elapsed, performing a maximum number of steps, or due to exiting the volume. Once the trajectory is calculated, it is then utilized by the flow visualization technique, whether it is simply to plot the trajectory (streamlines/pathlines) or for other purposes (e.g., stream surfaces, FTLE). Importantly, this means that research on efficient particle advection in turn informs performance of nearly all flow visualization techniques.

The computational challenges for particle advection can be significant. While some particle advection workloads require few particles and/or few steps, others can involve billions of particles and tens of thousands of steps, meaning trillions of advection steps overall. Further, the cases with many, many particles (billions) often come from computational simulations on supercomputers. Typically, these simulations have highly refined computational meshes (i.e., billions of cells or more). As a result, the vector field from these simulations are often so large that they cannot fit into the memory of a single node. Instead, the mesh (and thus the vector field) has to be decomposed into blocks, with each block small enough that it can fit into memory.

Parallelization is a key approach for solving computationally challenging particle advection problems. Unfortunately, decomposing data into blocks creates a significant parallelization challenge: getting the right particle and the right block on the same node at the same time.

The visualization community has introduced several parallel algorithms to address this challenge. That said, there has been no study to date that performs a comprehensive comparison of the which parallelization algorithm works best for which workloads. In this paper, we fill this gap, by evaluating and comparing the most popular parallel particle advection algorithms. We perform an empirical study — a "bake-off" — to answer the following research questions (RQ):

- **RQ1**: Which parallelization algorithm performs best for a given workload?

- **RQ2**: Why does a parallelization algorithm perform best for a given workload?
- **RQ3**: What are the unsolved problems in parallel particle advection? Are there any workloads that are difficult to balance using existing parallelization algorithms?

II. Related Works

There are two main approaches to parallelizing particle advection algorithms: 1) parallelizing-over-data (described in Section III-A1) and 2) parallelizing-over-particles (described in Section III-A2). All parallel particle advection algorithms involve either optimizing one of these two approaches, or using a hybrid approach involving both. The extensions for the parallelize-over-data approach include using round-robin block distribution [2] and using a pre-processing step [3]–[5]. These extensions focused on reducing the potential of load imbalance. The extensions for the parallelize-over-particles approach also include techniques to balance the workload, such as work requesting [6]–[8], and dynamic load balancing [9], [10]. Other extensions for parallelize-over-particles reduce the I/O cost by using techniques such as extending the memory hierarchy [11], and data prefetching [12], [13]. Hybrid solutions [7], [14] have been proposed to reduce load imbalance while reducing the I/O cost.

A separate line of inquiry has been on hybrid parallelism, i.e., continuing to use distributed-memory across nodes, but adding shared-memory parallelism within each node. Camp et al. [15] were the first to consider hybrid parallelism and streamlines, showing surprising benefits due to reduced communication costs. This was followed by additional works [16], [17] that considered the best hardware to solve a particle advection workload, CPU or GPU. Finally, Pugmire et al. [18] devised a platform portable method for shared-memory particle advection, within VTK-m [19].

While all of these works are important to our study, none sufficiently answer our research questions. In particular, many studies have incomplete comparators. As an example, our own work on Lifeline-based work requesting [8] extended parallelize-over-particles, and so compared to only parallelize-over-particles and other work requesting studies. It did *not* compare to parallelize-over-data techniques, hybrids between parallelize-over-data and parallelize-over-particles, or even optimizations for parallelize-over-particles that did not include work requesting.

The closest works to an empirical study have been:

- The algorithmic paper that introduced the Master/Worker approach [20], a hybrid between parallelize-over-data and parallelize-over-particles. Since it was a hybrid between the two approaches, it compared with both. That said, this study is over ten years old, leading to significant shortcomings: (1) it considers only modest scale and data size, (2) it does not consider algorithmic advancements that have occurred in the last decade, and (3) it was performed with "MPI-only" parallelism, since it predated research on hybrid parallelism for particle advection.
- The Camp et al. study [15] that demonstrated the value of hybrid parallelism for particle advection. Once again, this study has shortcomings with respect to our own research questions: (1) it considers runs of only 128 cores and (2) it looks at only parallelize-over-data and parallelize-over-particles.

In total, we feel there is no study that provides a holistic understanding of parallel particle advection for scientific visualization workloads on modern supercomputer platforms, and also that the closest previous works were pursuing different research questions. The purpose of this empirical study is to close this research gap, which in turn addresses the research questions identified in the introduction. Further, an important component to our study is that we implement all of the algorithms within the same software framework. This ensures fairer comparisons than often occur within studies that introduce a new algorithm and focus on workloads that best align with their new algorithm.

Finally, an important distinction for particle advection is whether or not the velocity field evaluations are for time-varying data. The large majority of parallelization techniques focus on a single time step, although some also consider time-varying data [2], [4], [21], [22]. In total, we view the topic of parallelization for time-varying to be at its infancy, and not yet deserving of inclusion in an empirical study. As a result, the scope of our study is for single time slice data. In our opinion, this choice does not diminish our study, as most particle advection use cases are also for single time slice data, in order to stave off exorbitant I/O costs.

III. Empirical Study Overview

This section provides an overview of our empirical study. It is organized into three parts: Subsection III-A describes the algorithms we consider, Subsection III-B describes our software system that implements these algorithms, and Subsection III-C describes considerations behind the factors that define our workloads.

A. Overview of Studied Algorithms

This subsection describes the four parallel particle advection algorithms considered in our study. We chose these four since they are some of the most used algorithms today, and also to consider a diversity in approaches. In summary, we study:

- The two main parallelization algorithms, parallelize-over-data and parallelize-over-particles, to understand the strengths and drawbacks of each one.
- A work requesting-based extension of parallelize-over-particles [6], to understand the tradeoff between the cost of the additional communication and load imbalance.
- The Master/Worker algorithm [20], which is a hybrid between parallelize-over-data and parallelize-over-particles, to understand their performance benefit, which is particularly important given their increased implementation cost.

In the remainder of this subsection, we briefly describe each of the four algorithms. Our descriptions are broken into two parts: overview of the algorithm and intuition about its performance characteristics.

*1) Parallelize-Over-Data Algorithm (**POD**):* In this method, the data is distributed among nodes. Each node advects the particles located in its data block until it terminates or exits the current block. When a particle leaves the current data block, it is communicated to the node that owns the needed data block. This process continues until all particles have reached termination.

This method reduces the I/O cost, which is usually more expensive than the computational cost. However, in the case of a dense seeding distribution (i.e., small seeding box) this method can suffer from load imbalance.

*2) Parallelize-Over-Particles Algorithm (**POP**):* In this method, particles are distributed across nodes. Particles are sorted spatially before distributing them to different nodes to enhance spatial locality. Each node advects its particles and load data blocks on demand. To reduce the I/O cost, this method advects all particles that are located in the current loaded data block until they terminate or exit the block. This method stores blocks in cache and uses the least-recently used (LRU) approach to remove data blocks from cache.

While this method has a better load balance in the case of a dense seed distribution, it has a high I/O cost.

*3) Work Requesting Algorithm with the Lifeline Scheduling Method (**LSM**):* An extension of POP that uses a dynamic load balancing method based on work requesting has been proposed by Mueller et al. [6]. The algorithm starts by distributing particles across nodes. Each node advects its particles and loads data on demand. When a node finishes its work, it requests work from other nodes. The requesting node is called a thief and the other node is called a victim. The most common approach to choose a victim is randomly [23]. In this paper, we use our improved work requesting algorithm that uses the Lifeline Scheduling Method [8] (LSM).

This method can suffer from high I/O and additional communication cost.

*4) Master/Worker Algorithm (**MW**):* Hybrid solutions have been proposed to address the limitations that POD and POP have individually and maintain load balance. Pugmire et al. [20] proposed a hybrid solution known as the master/worker. In this method, nodes are divided into groups, where each group has a master and workers. Particles are distributed equally among groups, and each master assigns particles to workers. Each node advects it is particles, when a node needs a data block, the master follows a set of rules to decide whether the worker should load the block or send the particle to another worker.

While hybrid solutions reduce load imbalance, they are more complicated to implement and they introduce additional costs.

B. Software System Overview

This section describes the details of our testing infrastructure. It is organized as follows: foundational algorithmic concepts (III-B1), carrying out advection work (III-B2), and communication between nodes (III-B3).

Algorithm 1 Skeleton pseudocode for the four algorithms.

```
1: numActive ← TotalNumParticles
2: activeParticles ← GenerateSeeds()
3: while numActive > 0 do
4:     if activeParticles.size() > 0 then
5:         WorkerFunction(activeParticles)
6:     end if
7:     CommunicationFunction(activeParticles)
8: end while
```

1) Foundational Algorithmic Concepts: All four algorithms described share common elements. First, they start by generating the seeds and distributing them among compute nodes. Then, in all four algorithms, each node executes the main loop which is composed of a worker function and a communication function. The worker function performs the I/O operations, the advection, and the processing of particles after each advection round. The communication function sends and receives data. This data can be particles or messages. The algorithm completes when all particle trajectories are calculated. Pseudocode 1 describes the general program that runs identically on each node for all four algorithms.

The pseudocode uses the following building blocks:

- GenerateSeeds(): a function that generates the initial seeds.
- WorkerFunction(): a function that performs I/O operations, advection and process the particles after advection.
- CommunicationFunction(): a function that sends and receives data (particles or messages) between nodes.

The implementation for these functions varies depending on the algorithm.

2) Worker Function: The worker function is responsible for executing three operations: 1) I/O, 2) advection, and 3) processing particles.

The I/O operation varies depending on the algorithm; it can be either a static allocation or load on demand. If the algorithm uses static allocation, then each node only reads the blocks assigned to it. If the algorithm uses load on demand, then each node loads the data blocks as needed. The POD algorithm uses static allocation, while the POP and LSM algorithms use load on demand. The MW varies, with workers using either load on demand or static allocation, based on instructions from the master.

In all four algorithms, each node passes the particles in the current data block to the VTK-m [18], [19] routine. The VTK-m routine performs the advection using shared-memory parallel. For our experiments, we used VTK-m's Intel Threading Building Blocks [24] option.

Finally, each node must perform particle processing after performing advection, and the four algorithms have both similarities and variations for this processing. For all four algorithms, each node terminates the particles that reached the maximum number of advection steps or exited the data set. Each node also notifies the other nodes of the number of terminated particles. The four algorithms vary in the way

they handle particles that exited the current data block. In the POD algorithm, the node stores the particle in a *communicate* queue to be sent to other nodes. In the POP and LSM algorithms, the node stores the particle in an *inactive*, which will be processed after advecting all the particles from the current block. In the MW algorithm, at each iteration, workers either communicate particles or store the particles in an *inactive* queue and load the needed data block.

3) Communication Function: The communication function is responsible for sending and receiving data, which can be particles or messages.

We built a communication routine that uses the Message Passing Interface (MPI) [25] for communication across the nodes. The routine uses a non-blocking communication, and can communicate messages and particles. It takes care of serializing and de-serializing the data.

Different algorithms communicate different types of data. In the four algorithms, each node sends a message to other nodes to notify them with the number of particles it terminated. Both POD and LSM communicate particles, the POD algorithm, nodes communicate particles according to the data block assignment, while in the LSM algorithm an idle node requests particles from other nodes. In the MW algorithm, nodes exchange different types of data. Master and workers exchange messages, where workers request work or a data block from the master. At the end of each iteration, the worker updates the master with information about its status, including the block IDs currently loaded and the number of particles it has. The master uses this information to send workers instructions. Workers communicate particles to other workers according to the master's instructions.

C. Workload Factors

An important consideration for an empirical study is what workloads to evaluate. In our case, we consider multiple factors that impact the particle advection workload. We then consider what happens as these factors vary, and how algorithm performance varies. In the remainder of this section, we describe the workload factors we consider. In Section IV (Experiment Overview), we describe specific options for each factor.

- **Size of Seeding Box**: The size of the seeding box represents the particle distribution. This impacts the I/O cost and can impact load balance. If the distribution is dense i.e., all the seeds originate in a small box, only a subset of the data set will be required, which reduces the cost of I/O. However, this might result in load imbalance if only a subset of the nodes is responsible for advecting these particles.
- **Total Number of Steps**: The total number of steps represents the amount of work defined as the product of the number of particles and the maximum number of advection steps. Different flow visualization algorithms require different representations. Some algorithms require a small number of particles that advect for a long duration, while others require a large number of particles that advect for a short duration. As the number of total advection steps increases, the computation complexity increases and thus distributing this complexity might be important. On the other hand, if that number is small, then it is better to distribute the data to reduce the I/O cost.
- **Concurrency**: Distributing the workload among multiple MPI ranks increases the amount of computational power and memory available. This can reduce the computation per rank, however, additional communications may also be required to better load balance the workload.

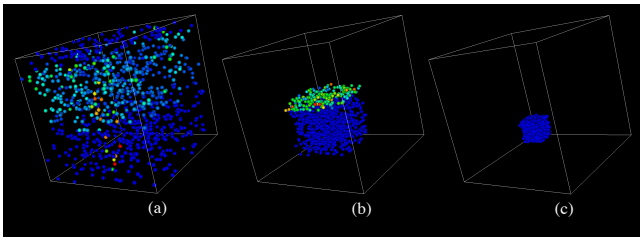

Fig. 1. The three seeding boxes considered in the study: (a) large, (b) mid, and (c) small.

Although the underlying vector field can clearly affect performance, we did not consider it as a workload factor. Many previous studies [8], [15], [17], [20] have considered multiple data sets. The intuition behind studying multiple data sets is to evaluate the extent that vector field changes performance. For example, a sink in the vector field can attract many particles to a single block, which can create imbalance for some algorithms. That said, these previous studies have consistently found underwhelming results with respect to the vector field significantly altering performance. Therefore, we decided to apply our limited compute resources to consider additional choices for the three workload factors we felt would most significantly inform this space.

IV. Experiment Overview

This section describes the details of our study.

A. Algorithm Comparison Factors

There are three main axes to our study:

- Size of seeding box (3 options)
- Total number of steps (6 options)
- Number of MPI ranks (3 options)

In total, we considered 54 (=3*6*3) configurations. We tested each configuration with all four algorithms, meaning 216 (=54*4) experiments overall. The following subsections discusses specific configurations.

1) Size of Seeding Box: In this study, we consider three options for the size of the seeding box: "large" box, "mid" box, and "small" box. The large box has particles uniformly spread through the data set, while the small box has them in a small sub-region. Figure 1 plots the three options.

2) Total Number of Steps: The total number of steps depends on the number of particles and the number of steps each particle goes (duration). We consider three options for the former, and two for the latter, for a total of 6 options. With respect to number of particles, we consider options as a ratio of particles per cell, i.e., 1 particle for each C cells. Our three options are: for every 100 cells, every 1000 cells, and every $10,000$ cells, and denote these as $P/100C$, $P/1KC$, and $P/10KC$, respectively ("$P/1KC$" meaning 1 particle for every one thousand cells). As the value of C decreases, the density of particles per cell increases, increasing the total number of particles. With respect to duration, we consider 1K and 10K steps.

Consider an example where the data set size is $1024*1024*512$. If we seed according to $P/10KC$, then we would have a particle for every $10,000$ cells. Since the total number of cells is approximately $537M$, then the number of particles will be approximately $54K$. Further, if the duration is 1000 steps, then the total number of advection steps would be $54M$ ($54K$ particles $\times$ 1000 steps per particle).

3) Concurrency: We test the weak scalability of the algorithms by varying the number of MPI tasks, as well as the number of data blocks. The runs were on the NERSC Cori machine (see Subsection IV-D), which has 32 cores per node. Each experiment placed 4 MPI tasks on a node, with 8 cores supporting each task. This enabled us to explore higher MPI task counts while staying within constraints on total cycles. The data increased with concurrency by adding additional blocks. At all concurrencies, the size of a given block was 128^3. In all, the experiments were as follows:

Config. Name	Nodes	MPI tasks	Cores	Blocks
Concurrency1	4	16	128	256
Concurrency2	32	128	1024	2048
Concurrency3	256	1024	8192	16384

Again, the goal was to inform the space while minimizing compute costs. Further, the tests were designed to increase in factors of eight, i.e., **Concurrency2** is $8X$ bigger than **Concurrency1** and **Concurrency3** is $8X$ bigger than **Concurrency2**.

B. Data Set

We use the "Fishtank" data set for our study, which comes from a thermal hydraulics simulation by the NEK5000 [26] code. In this particular simulation, twin inlets pump water of differing temperatures into a box, and the vector field captures the fluid flow within the box. The simulation's focus is on understanding mixing behavior, as well as temperature at the box's outlet.

C. Algorithm Settings

For all four algorithms, we use the Many-Core Visualization Toolkit (VTK-m) [18], [19] library for shared-memory parallelism. Many of the algorithms have "knobs" to optimizing their performance. In our study, we used the following values:

- **POP: Cache size:** The cache size in our study is 25 blocks per node, where each block has approximately two million cells. We adapted the settings presented by Camp et al. [11], while taking into account the data size and hardware differences between their study and ours.
- **LSM: Number of lifelines:** The number of lifelines was set as $log_2(\#Ranks)$, as per guidance in [8].
- **MW: Group size:** The group size varied depending on the number of MPI tasks (for details see Section III-A4). We tried different group sizes and found out that the best results are when there are 4 masters:
  - 16 Ranks: group size is 4, which means there are 4 masters.
  - 128 Ranks: group size is 32, which means there are 4 masters.
  - 1024 Ranks: group size is 256, which means there are 4 masters.

D. Hardware Used

The study was run on Cori at Lawrence Berkeley National Laboratory's NERSC facility. It contains 2,388 Intel Xeon "Haswell" processor nodes. Each node has two 2.3 GHz 16-core processors, each core supports 2 hyper-threads and there is 128 GB of memory per node.

E. Performance Measurements and Metric

For performance measurements, we instrumented our own timers using Unix system calls, capturing when each MPI rank was performing I/O, advection, communication, etc.

We also created a metric, $SPRPS$, to measure performance. Let S_T be the total number of advection steps for the workload, T be the total execution time for the slowest rank, and N be the number of ranks. Then we define $SPRPS$ as:

$$SPRPS = \frac{S_T}{(T * N)} \tag{1}$$

Intuitively, $SPRPS$ measures number of steps computed per rank per second. The higher the number of steps is, the more efficient the algorithm is, since it indicates lower execution time.

V. Data Corpus

This section introduces the data corpus generated by our experiments. This corpus is used in the analysis in Section VI, which addresses our research questions.

The main part of our corpus is Figure 2, which plots the efficiency of each algorithm with respect to our metric (*SPRPS*). The other part of our corpus is Figure 5, which plots the total execution time. As most of our analysis focuses on *SPRPS*, we place Figure 5 at the end of the paper. This enables other figures to be placed in a better flow within the paper, i.e., placing a figure and the analysis that incorporates the figure as proximate as possible.

We note two patterns from the organization of the figures:

- For two tests that have the same number of particles (for example $P/10KC * 1K$ and $P/10KC * 10K$), the performance is better when the duration is longer. This is because the total work done (advection step) per particle

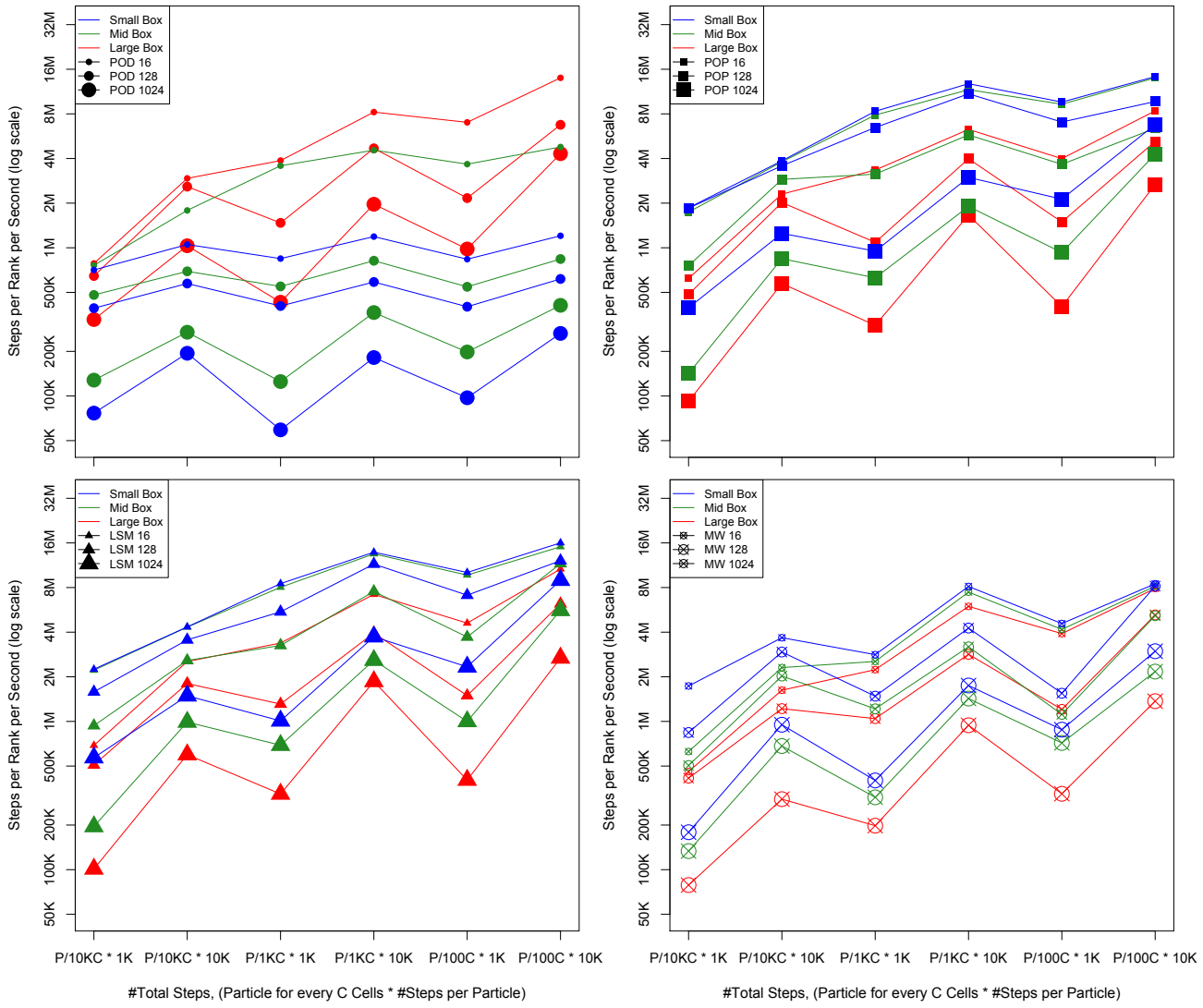

Fig. 2. Performance results for each of the four algorithms: POD (upper left), POP (upper right), LSM (lower left), and MW (lower right). For each sub-figure, the X axis is "Particle for every C Cells * #Steps per Particle." For example, $P/10KC*1K$ means there is one particle for every 10K cells and the duration for each particle is 1K advection steps. The leftmost value is the smallest number of total advection steps and the rightmost is the largest. However, some workloads are equal: 1) $P/10KC*10K$ is equal to $P/1KC*1K$ and 2) $P/1KC*10K$ is equal to $P/100C*1K$. The Y axis represents our performance metric: the number of steps per rank per second (see Section IV-E) — higher is better. Lines are colored by seeding size, and the size of each glyph corresponds to the concurrency.

is increased, which offsets the cost of the operations to manage particles.

- For two tests that have the same amount of work (total number of advection steps), the performance is better when the number of particles is smaller and the duration is larger. For example, $P/10KC*10K$, and $P/1KC*1K$ have the same amount of work, yet, the performance of $P/10KC*10K$ is better than $P/1KC*1K$. This is again due to amortizing costs for managing particles.

VI. Results

The results section is organized around our research questions — **RQ1** is addressed in subsection VI-A, **RQ2** in subsection VI-B, and **RQ3** in subsection VI-C.

A. *RQ1: Which parallelization algorithm performs best for a given workload?*

This question is relevant to both domain scientists and visualization practitioners. From a practical perspective, this question informs which algorithm to use in production settings.

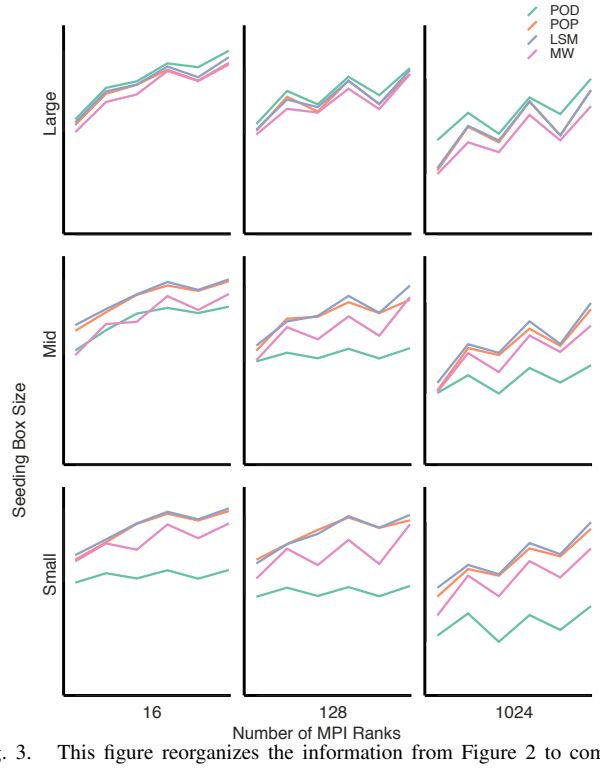

Fig. 3. This figure reorganizes the information from Figure 2 to compare algorithms by workload. Its nine subfigures are arranged as a 3x3 matrix, with concurrency changing across columns and seeding box size changing across rows. Each subfigure corresponds to a specific choice for concurrency and seeding box, showing the performance of all four algorithms. The algorithms are differentiated by color: POD is green, POP is orange, LSM is blue, and MW is pink. The units and plotting for each subfigure are the same as Figure 2. In particular, the X-axis has increasing number of steps, and the Y-axis is efficiency ($SPRPS$) in log scale.

TABLE I
$SPRPS$ FOR THE BEST PERFORMING ALGORITHM WITH RESPECT TO SIZE OF SEEDING BOX.

Size of Seeding Box		Small Box	Mid Box	Large Box
Best Algorithm		LSM	LSM	POD
Performance ($SPRPS$):	16 Ranks	16M	15M	14M
	128 Ranks	12M	11M	6M
	1024 Ranks	9M	5M	4M

Figure 3 addresses this research question by comparing the four parallelization algorithms. Each curve in Figure 3 also appears in Figure 2, with the difference being that Figure 3 is organized by workload, where Figure 2 is organized by algorithm. This new arrangement allows us to consider comparative performance (i.e., how the algorithms compares with respect to performance).

"Size of seeding box" is the only factor that influences comparative performance. If the seeding box is large, then POD offers the best performance. For the mid and small seeding boxes, LSM offers the best performance. That said, POP performance closely tracks LSM, making POP an attractive choice for this setting as well. Table I shows specifics about algorithm performance with respect to size of seeding box.

We find this result very surprising. Parallel particle advection has been well studied, and previous studies have shown

TABLE II
THIS TABLE CONTAINS INFORMATION ON THE BEST $SPRPS$ AND WORST $SPRPS$ FOR EACH OF THE FOUR ALGORITHMS, AND ALSO HOW THOSE EXTREME CASES SCALE. FOR EXAMPLE, THE HIGHEST $SPRPS$ FOR POD WAS ACHIEVED WITH THE LARGE SEEDING BOX AND THE $P/100C * 10K$ WORKLOAD, SO THIS WORKLOAD IS LISTED AS POD'S "BEST CASE." THE CHANGE COLUMN SHOWS THE DROP IN PERFORMANCE COMPARED TO THE SMALLEST CONCURRENCY (16 RANKS). FOR ALL FOUR ALGORITHMS, THE BEST $SPRPS$ WAS ACHIEVED WITH $P/100C * 10K$, WHICH WE ABBREVIATE "LW" (LARGEST WORKLOAD). SIMILARLY, THE WORST $SPRPS$ WAS ACHIEVED WITH $P/10KC * 1K$, WHICH WE ABBREVIATE "SW" (SMALLEST WORKLOAD).

POD Best Case:	**Large Box, LW**		
Concurrency	$SPRPS$	Time (s)	Change
16	14M	223	–
128	6.7M	462	2.1X
1024	4.3M	713	3.2X
POD Worst Case:	**Small Box, SW**		
Concurrency	$SPRPS$	Time (s)	Change
16	700K	4.41	–
128	319K	7.99	1.8X
1024	76K	40.6	8.7X
POP Best Case:	**Small Box, LW**		
Concurrency	$SPRPS$	Time (s)	Change
16	14.2M	219	–
128	9.6M	322	1.5X
1024	6.7M	432	2.1X
POP Worst Case:	**Large Box, SW**		
Concurrency	$SPRPS$	Time (s)	Change
16	600K	5.03	–
128	484K	6.44	1.3X
1024	92K	34.2	6.8X
LSM Best Case:	**Small Box, LW**		
Concurrency	$SPRPS$	Time (s)	Change
16	16M	195	–
128	12M	257	1.3X
1024	9M	326	1.7X
LSM Worst Case:	**Large Box, SW**		
Concurrency	$SPRPS$	Time (s)	Change
16	690K	4.53	–
128	508K	6.14	1.4X
1024	101K	30.8	6.8X
MW Best Case:	**Small Box, LW**		
Concurrency	$SPRPS$	Time (s)	Change
16	8.4M	368	–
128	8.1M	382	1.03X
1024	2.9M	984	2.6X
MW Worst Case:	**Large Box, SW**		
Concurrency	$SPRPS$	Time (s)	Change
16	458K	6.81	–
128	414K	7.54	1.1X
1024	78K	39.6	5.8X

that performance varies greatly based other factors as well, specifically total number of steps and concurrency. That said, our results do not overturn these previous studies: we also find that total number of steps and concurrency impact performance greatly. However, our novel contribution is in identifying that *comparative performance* is not affected. In other words, it is possible to select the best parallel algorithm based on seeding box, regardless of choices for the other factors.

Finally, the performance of the MW algorithm bears special discussion. For mid and small seeding boxes at 128 ranks, MW jumps in performance (relative to the other algorithms) at the most compute-heavy workload (P/100C * 10K). In the mid case, it outperforms POP, and, in the small case, it

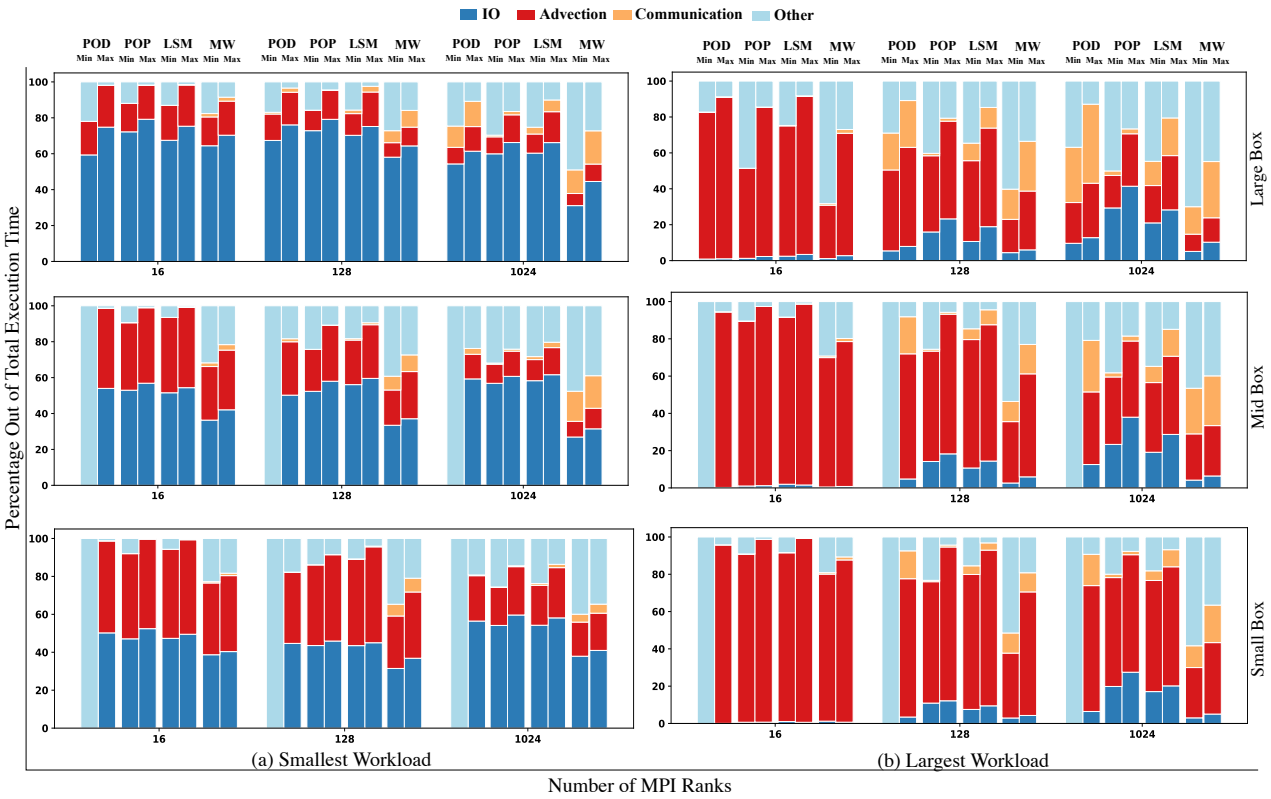

Fig. 4. Analyzing how algorithms spend their time, among four possible actions: I/O (dark blue), advection (red), communication (orange), and other, which includes idle time and management cost (light blue). The figure is broken into two sub-figures. Sub-figure (a), on the left, considers the workload with the least computation, i.e., those that involve $P/10KC * 1K$. Sub-figure (b), on the right, considers the workload with the most computation, i.e., those that involve $P/100C * 10K$. Each sub-figure contains three rows, with each row corresponding to a seeding box size (t-b: large, mid, small). Within a row, there are three groupings, with each grouping corresponding to a concurrency (l-r: 16 ranks, 128 ranks, 1024 ranks). A grouping consists of eight bars. These eight consist of the MPI Rank that does the lowest proportion of advection ("Min") and highest proportion of advection ("Max") for each of the four algorithms. The proportion of a bar devoted to a color indicates the percentage of time spent doing that activity. For example, if a bar is half red, then that MPI Rank spent 50% of its time doing advection.

is similar to POP. Extrapolating to even more computation-heavy work cases, it is conceivable that MW could eventually outperform either algorithm. That said, P/100C * 10K already likely represents a practical upper bound, as it represents ˜1000 seconds of execution time. Even more computationally-heavy workloads would likely be prohibitive, except for off-line scenarios.

*B. **RQ2**: Why does a parallelization algorithm perform best for a given workload?*

This question is relevant to visualization researchers, as it informs future algorithms that could maintain strengths while addressing weaknesses. Further, understanding this question provides further credibility to our conclusions for **RQ1**.

The performance of each algorithm is explored in its own subsection. That said, each subsection uses common information to inform its analysis. Specifically, Figure 4 shows breakdowns for how individual algorithms are spending their time (I/O, advection, communication, idle). Further, Table II shows performance for best and worst-case scenarios, as well as exploring effects due to changes in concurrency. Of course, our data corpus from Section V (i.e., Figures 2 and 5) is used as well.

1) POD Behavior: POD provides the most extreme performance of the four algorithms, with the best overall with the large seeding box and the worst overall for the mid and small seeding boxes.

In the large seeding box case, POD is able to keep MPI ranks busy enough to outperform the other algorithms. As seen in Table II, POD achieves an $SPRPS$ of 14M in its best configuration, which is competitive with the other algorithms in their best configurations. That said, the $SPRPS$ drops as the concurrency increases. Figure 4(b) explains why. At low concurrency, all MPI Ranks are performing advection nearly the entire time. Further, the load is balanced — the "Min" rank is performing nearly as much as advection as the "Max" rank. This balance is maintained as the problem is scaled to 1024 ranks, but the efficiency drops. Instead of the advection rates above 80% seen at 16 ranks, the rates drop to ˜30%, as communication plays an increasingly large role. This is because each block takes up a smaller portion of the spatial domain, and so particles cross blocks more often (leading to communication).

The mid and small seeding box cases expose POD's shortcomings. The "Min" ranks in all six relevant experiments

in Figure 4 are 100% idle. While Figure 4 shows only the behavior of the "Max" and "Min," our analysis of the other ranks shows that a large number are idle, and fully explain the slowdown. Ranks that are responsible for the data blocks inside the seeding box perform all the work, while other ranks remain idle during the entire execution time. In short, compute resources are not being used effectively.

2) POP Behavior: With POP, every MPI rank operates independently, and so each rank achieves similar efficiency. This similarity can be seen in all POP configurations in Figure 4, specifically with the "Min" profile being quite similar to "Max" profiles for each workload.

Table II and Figure 2 show that POP performs better on small boxes than large boxes. This reason is that a small seeding boxes starts more particles within the same block, so fewer block loads are needed overall. Figure 4 confirms this intuition. In particular, Figure 4(a) shows a trend where the proportion spent doing I/O decreases as the seeding box gets smaller. For 16 MPI Ranks, POP is spending nearly 80% of its time doing I/O for a large box, but only 50% of its time for a small box. The actual time to perform advection for these two workloads is the same, meaning that the small box larger proportion doing advection is indicative of a faster run-time.

POP's scalability is generally good for small box (dropping by a factor of 2X), but much poorer for large box (dropping by a factor of 6.8X). As blocks get smaller, particles cross block boundaries more often, leading to more I/O. That said, this effect is lessened for small box, since the particles being advected often lie in the same block.

3) LSM Behavior: LSM is generally similar to POP, and the trends described in Section VI-B2 are applicable to this algorithm as well. Figure 4 supports this, showing similar profiles for POP and LSM, although LSM has more communication from searching for additional work. Table II shows that LSM is a little better than POP for both "best case" and "worst case" and that they scale similarly. In short, the ability to improve workload balance via work requesting outweighs the overhead to locate victims.

4) MW Behavior: MW contained surprising findings. Despite MW outperforming POD and POP in a previous study [20], we found that MW did not perform well in our experiments. We note that a key difference was that the previous study, performed over ten years ago, did not use hybrid parallelism. Using hybrid parallelism increases the ratio between compute and communication. Further, our experiments had more compute per node than the previous study. Finally, Figure 4 shows that MW had a significant amount of idle time — the "Max" MPI Rank was waiting for work at least 20% of the time in all of our configurations.

Despite this, MW does have merit. Figure 3 shows that MW beats POP for the workload consisting of $P/100C*10K$ ("Largest Workload") with 128 MPI Ranks and the Mid box. Looking at Figure 4(b), MW is doing much less I/O than POP, offsetting its idle time. Improvements to MW that account for increased compute per node could possibly propel the algorithm to better performance than the other algorithms.

Finally, Table II shows that MW scaled quite poorly with the large seeding box.

C. *RQ3: What are the unsolved problems in parallel particle advection? Are there any workloads that are difficult to balance using existing parallelization algorithms?*

These questions are relevant to both visualization researchers and domain scientists. For visualization researchers, these questions inform research gaps for future algorithmic improvements. For domain scientists, these questions inform workloads to avoid when using visualization tools.

Our treatment of **RQ3** is divided into two parts, due to the nature of our study. Our empirical study selected four algorithms for evaluation, since these algorithms are popular today. That said, those four algorithms do not represent all possible algorithms, and so our answers for **RQ3** must reflect that our results are for a subset of algorithms. We do this by answering **RQ3** in two ways. Subsection VI-C1 considers how the findings from this study inform **RQ3**. Subsection VI-C2 considers additional algorithms that were not part of the study, and speculates on their role in answering **RQ3**.

*1) Findings from This Study for **RQ3**:* This subsection answers **RQ3** from the perspective of the four studied algorithms. We see three main findings:

- **F1**: Execution times are quite slow on workloads with a large number of total steps. That said, these times are in-line with previous findings.
- **F2**: Scalability is acceptable, but not excellent. Our experiments rose in concurrency from 16 MPI Ranks to 1024 MPI Ranks (128 cores to 8192 cores), for an increase of 64X. Since we ran a weak scaling study, we would expect execution time to remain constant as concurrency increased. Instead, we saw execution times go up by a factor of 2X to 4X, depending on workload. (Note that some algorithms worsened by more than 2X or 4X; the preceding sentence is referring to the behavior of the best performing algorithm for a workload, i.e., POD for large seeding boxes and LSM for the remainder.)
- **F3**: Efficiency is much lower for workloads with low numbers of advection steps. Figure 4(a) shows that I/O is the main reason — there is not enough advection work to amortize loading of blocks.

*2) Speculation on **RQ3** Regarding Additional Algorithms:* In this section, we speculate on algorithms that may address some of the findings above:

- Regarding finding **F2**, Peterka et al. [2] extended POD to use a round-robin assignment and dynamic geometric repartitioning. Importantly, they were able to achieve better scalability with a large seeding box.
- Regarding finding **F3**, improving efficiency would need to come through reduced I/O costs. For POP, LSM, and MW, one possibility would be to use more of a deep memory hierarchy to prevent reloading blocks. This was first suggested by Camp et al. [11] for POP. That said, finding **F3** centers around small workloads, so it is unlikely that

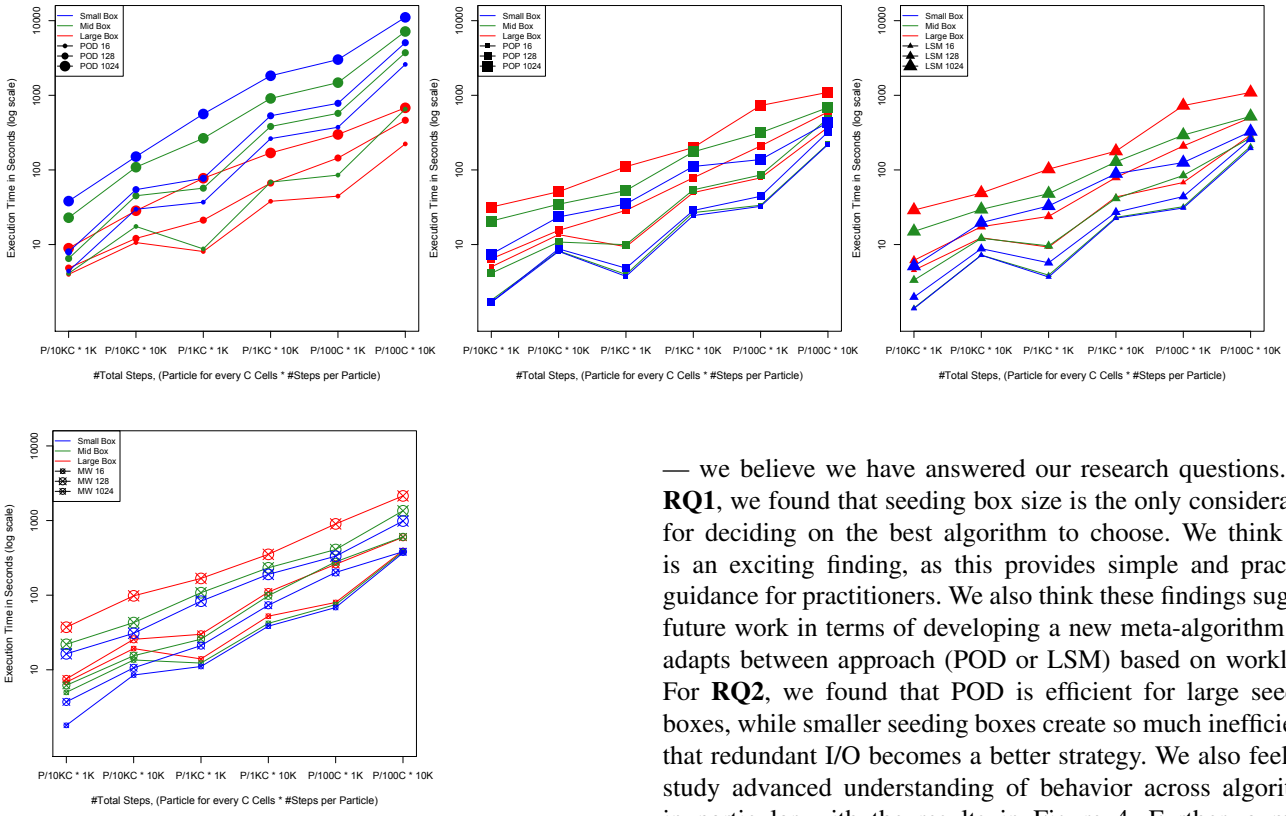

Fig. 5. Execution times for our four algorithms: POD (top left), POP (top middle), LSM (top right), and MW (bottom). The organization matches Figure 2, except the Y-axis is execution time (log scale) — lower is better.

blocks will be reloaded repeatedly. Another idea is to fetch blocks from other MPI Ranks rather than the disk, with the thinking that communication will be faster than I/O. This direction was first suggested by Peterka et al. [7]. That said, we implemented this direction for our own study, and found that it did not affect performance (disk loads were as fast as retrieving from other MPI Ranks). As a result, we used a straightforward POP implementation for our study.

Revisiting our findings from Section VI-C1, we feel only finding **F2** requires additional scrutiny. From our own experiments, we felt scalability was acceptable, although not perfect. Our worst performing experiments were large seeding boxes, and we speculating that the round-robin assignment with dynamic geometric repartitioning would improve scalability for this case. Assuming it could boost performance by 2X at large scale, then best performing algorithms would be achieving ˜2X slowdowns over 64X increases in concurrency.

VII. Conclusion and Future Work

Our empirical study was designed to answer three research questions. While a fixed compute allocation bounded the number of experiments we could perform, we were pleased that our experiments provided significant clarity for our questions — we believe we have answered our research questions. For **RQ1**, we found that seeding box size is the only consideration for deciding on the best algorithm to choose. We think this is an exciting finding, as this provides simple and practical guidance for practitioners. We also think these findings suggest future work in terms of developing a new meta-algorithm that adapts between approach (POD or LSM) based on workload. For **RQ2**, we found that POD is efficient for large seeding boxes, while smaller seeding boxes create so much inefficiency that redundant I/O becomes a better strategy. We also feel our study advanced understanding of behavior across algorithm, in particular with the results in Figure 4. Further, a major finding was that the MW algorithm was performing poorly, suggesting that future work should look at updating MW's rules and procedures for today's increased compute capability on a node. Finally, for **RQ3**, we see that execution times on large workloads are quite large. While consistent with previous studies, our study increases the evidence that this issue needs more attention, especially since these workloads enable deeper understanding of flow fields, such as Lagrangian Coherent Structures. In particular, the computer graphics ray tracing community has invested significantly in preprocessing steps and acceleration strategies (hot caches, ray bundling) that enable billions of ray intersections in seconds. When asked to perform billions of advection steps, we speculate that similar preprocessing and acceleration strategies could benefit particle advection.

Future work could explore the limitations of our study that stemmed from having finite compute cycles. In particular, we ran on the NERSC Cori machine, which fixes the ratios between compute, I/O, and communication. Certainly, other machines would see different execution times. That said, we predict that our findings for **RQ1** and **RQ2** will translate to new environments, since they speak to load balancing issues. However, GPU-based supercomputers may have better execution times (**RQ3**), partially addressing this issue. Finally, limits in available compute time forced us to consider only one data set and block size. While we do not expect these factors to significantly alter the answers to our research questions, future work could confirm this.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was supported by the Scientific Discovery through Advanced Computing (SciDAC) program of the U.S. Department of Energy. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] P. Prince and J. Dormand, "High order embedded Runge-Kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 7, no. 1, pp. 67 – 75, 1981.

[2] T. Peterka, R. 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 *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 580–591, May 2011.

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

[4] H. Yu, C. Wang, and K. Ma, "Parallel hierarchical visualization of large time-varying 3d vector fields," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, Nov 2007.

[5] B. Nouanesengsy, T. Y. Lee, and H. W. Shen, "Load-balanced parallel streamline generation on large scale vector fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 1785–1794, Dec 2011.

[6] C. Müller, D. Camp, B. Hentschel, and C. Garth, "Distributed parallel particle advection using work requesting," in *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 1–6, Oct 2013.

[7] K. Lu, H. Shen, and T. Peterka, "Scalable computation of stream surfaces on large scale vector fields," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1008–1019, Nov 2014.

[8] R. Binyahib, D. Pugmire, B. Norris, and H. Childs, "A lifeline-based approach for work requesting and parallel particle advection," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 52–61, Oct 2019.

[9] D. Morozov and T. Peterka, "Efficient delaunay tessellation through k-d tree decomposition," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 728–738, Nov 2016.

[10] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka, "Dynamic load balancing based on constrained k-d tree decomposition for parallel particle tracing," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, pp. 954–963, Jan 2018.

[11] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. I. Joy, "Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 57–64, Oct 2011.

[12] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr, "Iteration aware prefetching for large multidimensional datasets," in *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, SSDBM, (Berkeley, CA, US), pp. 45–54, Lawrence Berkeley Laboratory, 2005.

[13] O. O. Akande and P. J. Rhodes, "Iteration aware prefetching for unstructured grids," in *IEEE International Conference on Big Data*, pp. 219–227, Oct 2013.

[14] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson, "Simplified parallel domain traversal," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 10:1–10:11, ACM, 2011.

[15] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy, "Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 17, pp. 1702–1713, Nov. 2011.

[16] 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 *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, (Girona, Spain), pp. 1–8, May 2013.

[17] H. Childs, S. Biersdorff, D. Poliakoff, D. Camp, and A. D. Malony, "Particle Advection Performance over Varied Architectures and Workloads," in *IEEE International Conference on High Performance Computing (HiPC)*, (Goa, India), pp. 1–10, Dec. 2014.

[18] D. Pugmire, A. Yenpure, M. Kim, J. Kress, R. Maynard, H. Childs, and B. Hentschel, "Performance-Portable Particle Advection with VTK-m," in *Eurographics Symposium on Parallel Graphics and Visualization*, The Eurographics Association, 2018.

[19] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications (CG&A)*, vol. 36, pp. 48–58, May/June 2016.

[20] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber, "Scalable computation of streamlines on very large datasets," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, Nov 2009.

[21] C. Chen, B. Nouanesengsy, T. Lee, and H. Shen, "Flow-guided file layout for out-of-core pathline computation," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 109–112, Oct 2012.

[22] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 61:1–61:11, IEEE Computer Society Press, 2012.

[23] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[24] Intel Corporation, "Introducing the Intel Threading Building Blocks," May 2017. https://software.intel.com/en-us/node/506042.

[25] "Mpi: A message passing interface," in *Supercomputing '93. Proceedings*, pp. 878–883, Nov 1993.

[26] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, "nek5000 Web page," 2008. http://nek5000.mcs.anl.gov.