A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection

Roba Binyahib* University of Oregon
David Pugmire† Oak Ridge National Laboratory
Boyana Norris‡ University of Oregon
Hank Childs§ University of Oregon

ABSTRACT

Particle advection, a fundamental building block for many flow visualization algorithms, is very difficult to parallelize efficiently. That said, work requesting is a promising technique to improve parallel performance for particle advection. With this work, we introduce a new work requesting-based method which uses the Lifeline scheduling method. To evaluate the impact of this new algorithm, we ran 92 experiments, running at concurrencies as high as 8192 cores, data sets as large as 17 billion cells, and as many as 16 million particles, comparing against other work requesting scheduling methods. Overall, our results show that Lifeline has significantly less idle time than other approaches, since it reduces the number of failed attempts to request work.

Index Terms: Work requesting—Visualization—Parallel particle advection—Lifeline-Based scheduling

1 INTRODUCTION

Flow visualization techniques are used to understand the behavior and patterns occurring in vector fields. There are a variety of techniques that are used in practice. Examples include streamlines, pathlines, stream surfaces, Poincaré analysis, and Finite-Time Lyapunov Exponents (FTLE). These flow visualization techniques are built upon particle advection, which consists of placing a particle in the flow and displacing its position according to a vector field. The trajectory of each particle as it is displaced can be described by an ordinary differential equation. In practice, this trajectory is calculated iteratively. A particle with an initial (seed) location, X_0, is displaced to a nearby location, X_1, then displaced again to another location, X_2, and so on. Each advancement, i.e., from X_i to X_{i+1}, is referred to as a step. Calculating the change in position for a given step is typically approximated using numerical methods, such as a Runge-Kutta. These numerical methods require a number of vector field evaluations at different spatial locations. Each visualization technique will then use the computed trajectories of each particle to display its representation of features within the flow field. The representations used by each technique are varied. For streamlines and pathlines the representations are simple, namely displaying the trajectory of each particle. For FTLE, it is more complex, as the technique creates derived quantities based on distances between trajectory end points.

Because each flow visualization technique requires different representations, the corresponding particle advection workloads are similarly highly varied. For example, when using a streamline technique, as few as one particle can be used. For another example, when computing an FTLE, one or more particles can be placed in each cell of a mesh, resulting in potentially billions of particles being used. Further, the number of steps can vary as well, from under one hundred steps to hundreds of thousands of steps, or more (e.g., Poincaré analysis [12, 20]). As a result, some particle advection workloads have excessive computation times — billions of steps, with each step requiring velocity field interpolations to solve an ordinary differential equation (Runge-Kutta).

Particle advection solutions get considerably more complex in the context of supercomputers. This setting typically adds two significant complications: (1) data sets contain many cells and are decomposed into blocks, and (2) the number of advection steps to calculate is so large that parallel processing is required. Supercomputer architectures add to the complexity as well, as they are made up of many nodes, with each node containing its own (private) memory. Ultimately, the fundamental challenge of efficient parallel particle advection on supercomputers is to make sure the correct particle and vector field information are together at the same time so a step can occur. Unfortunately, data set sizes often preclude the simplest method for achieving this solution — loading all vector field information on all nodes.

While there are multiple parallelization strategies, this paper focuses on the strategy where the fundamental unit of parallelization is a particle. In other words, the particles are partitioned over the supercomputer's nodes and each node calculates the trajectories of its particles. In the simplest form, often referred to as parallelize-over-particles (POP), a node loads blocks from disk as needed, and purges the blocks it has already read in when it runs low on memory. One pitfall for POP is that it suffers from idle time when some nodes finish their calculations before others.

An important optimization for POP is to incorporate work requesting. Work requesting is designed to minimize idle time — nodes that finish their calculations communicate with others nodes and request that they share some of their work. For example, if node N_i has finished its calculations and node N_j has to calculate the trajectories of P particles, then N_j can send $\frac{P}{2}$ particles to N_i, to even the load. Work requesting incorporates an underlying scheduling method, and previous work has incorporated the Random Scheduling Method (RSM) [11, 14].

With this work, we introduce a new algorithm for work requesting parallel particle advection. Our improvement is to incorporate the Lifeline scheduling method (LSM). LSM is currently the high-performance computing community's preferred scheduling method for work requesting [10, 21, 25]. Our findings show that, for parallel particle advection, LSM is superior to RSM in all cases, and reduces inefficiency by significant amounts. Finally, since we discovered that RSM has some fundamental limitations for particle advection problems, we also introduce an extension to RSM to request work from multiple victims, which we refer to as RSM-N (N victims). That said, we find with our experiments that RSM-5 (5 victims) is also inferior to LSM.

2 RELATED WORK

2.1 Parallel Particle Advection

There are multiple parallel particle advection techniques [18]. The parallelize-over-data (POD) method complements POP — POD divides blocks over nodes of a supercomputer and communicates

*e-mail: roba@cs.uoregon.edu
†e-mail: pugmire@ornl.gov
‡e-mail: norris@cs.uoregon.edu
§e-mail: hank@uoregon.edu

particles between nodes as they travel from block to block. Several extensions of POD have been presented to reduce the potential of imbalance. These studies used different solutions such as round-robin block assignment [16] and pre-processing [5, 15, 24]. Other studies proposed hybrid algorithms [9, 17], where the algorithm used both parallelization techniques (POP and POD) to balance the workload.

There have also been studies that look at how to make use of the shared-memory parallelism within each supercomputer node. Camp et al. [3, 4, 6] ran a series of studies showing the benefits of hybrid parallelism (i.e., using both distributed- and shared-memory on supercomputers) for particle advection on CPUs and GPUs. Their CPU-only studies [3] demonstrated speedups of up to 4X by having fewer communications and increasing workload within a node. These results have encouraged us to use hybrid parallelism in our own study. In particular, we use the VTK-m [13] solution for particle advection [19] within a node, and have custom MPI communication code for distributed-memory parallelism. In a separate study [2], Camp looked specifically at POP, and evaluated how POP performance could be improved by utilizing deep memory hierarchies to store more blocks (and thus reduce purges that lead to redundant loading of blocks). This study was useful for us in setting the parameters for setting the cache size, i.e., the number of blocks each node can store.

The most relevant previous works to our study are those that also consider work requesting to accelerate parallel particle advection. Mueller et al. [14] considered the technique for streamlines, while Lu et al. [11] considered the technique for stream surfaces. The key difference between their works and our own is that they employ RSM [1] while we employ LSM [21]. That said, the work by Mueller et al. extended RSM to have threaded communication.

2.2 Lifeline-Based Scheduling

Previous parallel particle advection solutions [11, 14], have used work requesting to improve parallel performance for particle advection. In work requesting, once a node finishes advecting its particles, it requests particles from another busy node. The node stealing the particles is called a thief and the other node is called a victim. These previous solutions used random scheduling [1]. Several works in the high-performance computing community showed improved performance over random using a Lifeline-based scheduling method [10, 21, 25], which was introduced by Saraswat et al. [21]. In our algorithm, we replace the traditional random scheduling with the lifeline scheduling method.

The Lifeline approach begins similarly to the random scheduling method, in that the thief node attempts some number of random steals, w. But the lifeline algorithm differs in how to proceed if the first w steals all fail (i.e., did not result in work being returned from the victim, because the victim also has no work). Instead, the node consults its lifelines (i.e., a list of compute nodes) to ask for work. What differentiates a lifeline steal from a regular steal is that lifeline is then engaged on behalf of the thief to find work. Each of the lifelines will store the thief as an "incoming" lifeline. When those lifelines search for work themselves, they will share the work with the thief.

The key to the Lifeline approach is the "Lifeline graph," which directs a thief node to use specific nodes as lifelines.

The lifeline graph is a fully-connected directed graph, where graph vertices are compute nodes on the supercomputer and edges are lifelines. This graph must guarantee that there is a path from each node with work to all other nodes. The simplest way to create one lifeline for each node is to create a circular graph where the lifeline of the rank ID p is $(p+1)\%N$, with N the number of ranks. This simple method is not acceptable in practice, though, since it will result in poor performance at scale. This is because the distance between two nodes is on average $\frac{N}{2}$ with N the total number of nodes. This means that requesting work to a victim would require on average $\frac{N}{2}$ communications, which is inefficient.

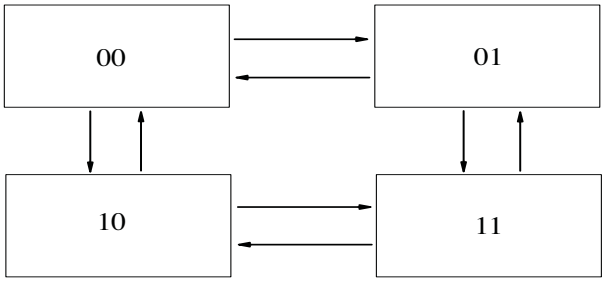

Figure 1: A lifeline graph of 4 nodes, with base = 2 and power = 2. Each node is represented in a base of 2 and has two lifelines. For each node, the outgoing arrows points to its lifelines.

Instead, the Lifeline algorithm used a cyclic hypercubes graph to calculate the lifelines. This guarantees that the graph is connected, has a low diameter, and each vertex has a bound on the number of out edges. To calculate the lifelines of nodes, the user has to choose a base h and a power z, with the constraint $h^{z-1} < N \leq h^z$, where N is the number of compute nodes. Each node is represented as a number in base h with z digits, and has an outgoing edge to every node that is distance +1 from it in the Manhattan distance. Figure 1 shows a lifeline graph for of four nodes, with base $h = 2$, and power $z = 2$, each node has two lifelines. Full details on the method can be found in the paper by Saraswat et al. [21]

3 Our Lifeline Algorithm

This section describes our lifeline-based algorithm, as well as other algorithms we compare against. It is organized as follows:

- Section 3.1 describes foundational concepts.
- Section 3.2 describes two existing algorithms — POP and RSM.
- Section 3.3 describes our LSM algorithm.
- Section 3.4 describes an extension to RSM to include more victims (RSM-N); this algorithm allows us to evaluate lifeline better.

3.1 Foundational Algorithmic Concepts

All four algorithms described share common elements. First, they each begin by dividing the set of P seed points over its N compute nodes, giving each node $\frac{P}{N}$ seed points to operate on. Each node then executes the same program, differing only in the seed points they begin with, and the algorithm completes when all particle trajectories are calculated. In the sections that follow, the pseudocode listed describes the program that runs identically on each node.

The pseudocode for our four algorithms use the following building blocks:

- Particle: a data structure that contains the particle to be advected through the flow field. This data structure contains the current location of the particle. It can optionally hold the previous locations of the particle (i.e., the trajectory) .
- ParticleArray: a data structure that contains an array of Particles.
- ArrayOfParticleArrays: a data structure that contains an array of ParticleArrays. For example, an ArrayOfParticleArrays with 10 entries would contain 10 ParticleArrays, with each of the 10 ParticleArrays containing a varying number of particles.
- SortParticleByBlock(): a function that sorts Particles by the ID of the block that contains the particles. This generates

Algorithm 1 Pseudocode for the Parallelize-Over-Particles algorithm (POP).

```
1:  function POP-ADVECT(ParticleArray pv)
2:      keepGoing ← true
3:      ArrayOfParticleArrays pva[NUMBLOCKS]
4:      pva ← SortParticlesByBlock(pv)
5:      allCompletedParticles ← ∅
6:      while keepGoing do
7:          contParticles ← ∅
8:          for i in NUMBLOCKS do
9:              if pva[i].size() > 0 then
10:                 Block b ← ObtainBlock(i)
11:                 ParticleArray completed, continuing
12:                 (completed, continuing) ← Advect(pva[i], b)
13:                 allCompletedParticles += completed
14:                 contParticles += continuing
15:             end if
16:         end for
17:         if contParticles.size() > 0 then
18:             pva ← SortParticlesByBlock(contParticles)
19:         else
20:             keepGoing ← false
21:         end if
22:     end while
23: end function
```

an ArrayOfParticleArrays where the ParticleArray at index *i* contains the Particles that lie within block *i*.

- ObtainBlock(): a function that determines the needed block and reads it from cache or disk. The function first checks if the block is already available in cache. If not, it loads the necessary block and places it in cache. The size of the cache changes depending on the size of the data.
- Advect(): a function that advects the Particles of a ParticleArray until they exit the current block or terminate. This function returns a 2-tuple — the first element is a ParticleArray containing completed Particles and the second element is a ParticleArray containing Particles that exited the current data block.
- CheckForIncomingMessages(): a function that checks for incoming messages from other nodes. These messages can be work requests from other nodes or notifications of particle terminations.
- SendWork(): a function that sends half of its workload to the thief if it has any work, or sends back a "no work" message.
- RequestWork(): a function that requests work from another node.

3.2 Existing Algorithms

This section describes two existing algorithms used as comparators for our study: POP and Work Requesting using the RSM.

3.2.1 Parallelize-Over-Particles

Algorithm 1 shows the pseudocode for POP. The algorithm starts by sorting particles by block. Then it reads the needed data blocks either from cache or disk. Next, the algorithm advects the particles located in the current data block until they terminate or exit the current block. When particles exit their current data blocks, they are stored in an array to be processed in the next iteration.

Even though the algorithm divides seeds equally between nodes, it does not guarantee an equal workload on each node. That is because nodes might load different number of blocks or have different number of advection steps, due to the nature of the vector field and placement of assigned seeds. For example, if the vector field has

Algorithm 2 Pseudocode for the working requesting algorithm using RSM.

```
1:  function RSM-ADVECT(ParticleArray pv)
2:      keepGoing ← true
3:      numActive ← totalNumberOfParticles
4:      ArrayOfParticleArrays pva[NUMBLOCKS]
5:      pva ← SortParticlesByBlock(pv)
6:      allCompletedParticles ← ∅
7:      while keepGoing do
8:          contParticles ← ∅
9:          for i in NUMBLOCKS do
10:             if pva[i].size() > 0 then
11:                 Block b ← ObtainBlock(i)
12:                 ParticleArray completed, continuing
13:                 (completed, continuing) ← Advect(pva[i], b)
14:                 allCompletedParticles += completed
15:                 contParticles += continuing
16:             end if
17:             MSG ← CheckForIncomingMessages()
18:             if MSG = PARTICLES_TERMINATED then
19:                 numActive -= MSG.numTerminated
20:             else if MSG = NEEDWORK then
21:                 SendWork()
22:             end if
23:         end for
24:         if contParticles.size() > 0 then
25:             pva ← SortParticlesByBlock(contParticles)
26:         else if numActive > 0 & contParticles.size() = 0 then
27:             randomVictim ← GetRandomVictimID()
28:             RequestWork(randomVictim)
29:         else
30:             keepGoing ← false
31:         end if
32:     end while
33: end function
```

critical points attracting the particles toward them, the workload of the node depends on the placement of its assigned seeds. Nodes that have seeds located near the critical points will need fewer block than particles that are far from the critical point.

3.2.2 Work Requesting using the Random Scheduling Method

Algorithm 2 shows the pseudocode of RSM. The algorithm begins with each node executing the POP algorithm as described in Section 3.2.1.

The algorithm is different, however, in how it proceeds when a node finishes its work. In this case, it sends a work request to another node. Again, the node stealing the particles is referred to as thief and the other node is referred to as victim. RSM chooses a victim randomly [1]. If the victim has work, it sends half of its workload to the thief. Otherwise, it sends a "no work" message to the thief. In that case, the thief selects another victim randomly.

To optimize I/O, the algorithm sorts particles by block before sending work. This reduces the number of blocks that need to be accessed.

3.3 Our Lifeline-Based Algorithm

This section describes our particle advection Lifeline-Based algorithm. Algorithm 3 shows the pseudocode of LSM. LSM shares most of the steps of RSM, with the main difference between them being the scheduling method.

In this algorithm, the thief performs *w* random steals, where the victims are chosen randomly, and *w* is a user specified parameter. If no work is found after *w* attempts, the thief requests work from its

Algorithm 3 Pseudocode for the working requesting algorithm using Lifeline Scheduling (LSM).

```
1:  function LSM-ADVECT(ParticleArray pv)
2:      keepGoing ← true
3:      numActive ← totalNumberOfParticles
4:      ArrayOfParticleArrays pva[NUMBLOCKS]
5:      pva ← SortParticlesByBlock(pv)
6:      allCompletedParticles ← ∅
7:      lifelines ← CalculateLifelineGraph()
8:      numRandomReq ← 0
9:      while keepGoing do
10:         contParticles ← ∅
11:         for i in NUMBLOCKS do
12:             if pva[i].size() > 0 then
13:                 Block b ← ObtainBlock(i)
14:                 ParticleArray completed, continuing
15:                 (completed, continuing) ← Advect(pva[i], b)
16:                 allCompletedParticles += completed
17:                 contParticles += continuing
18:             end if
19:             MSG ← CheckForIncomingMessages()
20:             if MSG = PARTICLES_TERMINATED then
21:                 numActive -= MSG.numTerminated
22:             else if MSG = NEEDWORK then
23:                 SendWork()
24:             end if
25:         end for
26:         if contParticles.size() > 0 then
27:             pva ← SortParticlesByBlock(contParticles)
28:         else if numActive > 0 & contParticles.size() = 0 then
29:             if numRandomReq < w then
30:                 randomVictim ← GetRandomVictimID()
31:                 RequestWork(randomVictim)
32:                 numRandomReq++
33:             else
34:                 RequestWork(lifelines)
35:             end if
36:         else
37:             keepGoing ← false
38:         end if
39:     end while
40: end function
```

lifelines. The lifelines are computed using a lifeline graph following the rules mentioned in Section 2.2. If the victim does not have any work, it requests work for its lifelines recursively. After the thief requests the work from its lifelines, it remains idle.

When a node receives work, it checks for incoming lifelines; if it has any, then it sends work.

Similar to RSM, the algorithm sorts its particles by block before dividing the workload among lifelines, to reduce I/O cost.

The number of lifelines for each node impacts the performance of LSM. If the number of lifelines is small, it might lead to a higher idle time. On the other hand, if the number of lifelines is large, it might increase the communication cost.

3.4 RSM-N: Extending RSM For Multiple Victims

For evaluation purposes, we also made a straightforward extension to the RSM algorithm, namely, to request work from multiple victims.

To conduct a fair comparison between the two scheduling methods, we adapted RSM to allow the thief to request work from the same number of victims as LSM. If no work is found, the thief chooses a new group of random victims.

4 EXPERIMENTS

This section describes the details, which compares the four algorithms described in Section 3: POP, RSM, RSM-N, and LSM. The additional factors considered in this study are described in the following subsection.

4.1 Algorithm Comparison Factors

Our study is composed of seven phases. The first phase considers one workload in depth, comparing the four algorithms. In the other six phases, one of six factors is varied, while holding the other five constant. The six factors are:

- Data set (4 options)
- Number of particles (4 options)
- Maximum advection steps (i.e., duration of particle) (4 options)
- Number of blocks (5 options)
- Number of cells per block (3 options)
- Number of MPI tasks (3 options)

In total, we considered 23 (= 4 + 4 + 4 + 5 + 3 + 3) configurations. We tested each configuration with all four algorithms, meaning 92 experiments overall.

4.1.1 Data Set

Since the complexity of the vector field impacts the performance, we test the performance of our algorithms on different data sets that broadly represent typical application scenarios. The four data sets used in this study are:

- The Fishtank data set is a thermal hydraulics simulation using the NEK5000 [8] code. In this particular simulation, twin inlets pump water of differing temperatures into a box. The mixing behavior and temperature of the water at the outlet of the box are of interest. The vector field captures the fluid flow within the box.
- The Fusion data set is a magnetically confined plasma in a tokamak device. The simulation was performed using the NIMROD [23] code. The vector field in this example is of the magnetic field that exists inside the plasma that is a result of the magnets in the tokamak device as well as the motion of the particles within the plasma itself.
- The Astro data set is the magnetic field surrounding a solar core collapse resulting in a supernova. This simulation was performed with the GenASiS [7] code, a multi-physics code for astrophysical systems involving nuclear matter.
- The RadialExpansion data set is an artificial dataset, where the vector for each point is measured by the distance from the location of the point to the center. The dataset was created to test the behavior of the four algorithms in cases where the load is highly imbalanced.

The four data sets are 3D steady data sets that were refined to a 1024^3 grid.

4.1.2 Number of Particles

With this factor, we consider the impact of the number of particles on the inefficiency of the four algorithms. Four amounts of particles are considered: 10K, 100K, 1M, and 10M.

4.1.3 Maximum Advection Steps

With this factor, we consider the impact of the advection steps on the inefficiency of the four algorithms. Four amounts of advection steps are considered: 100, 1K, 10K, and 100K.

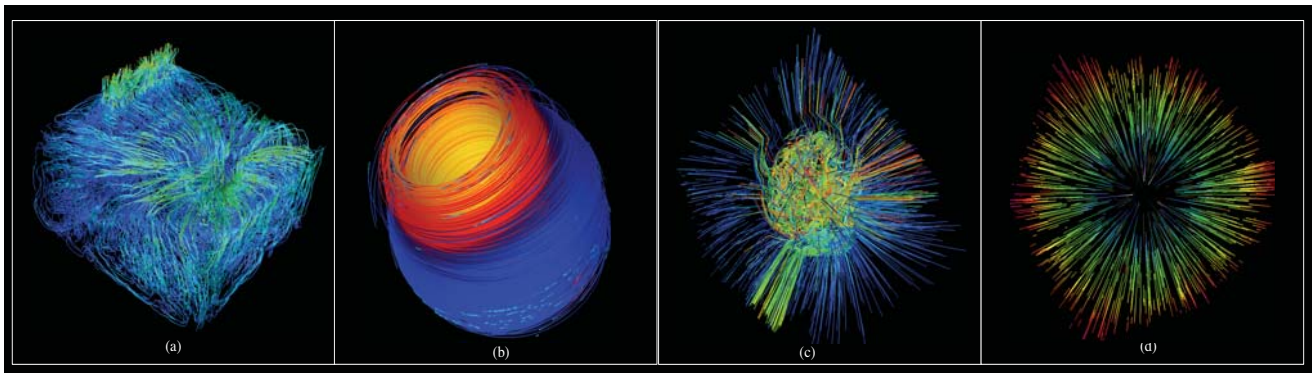

Figure 2: Streamlines visualization for the four data sets: (a) Fishtank, (b) Fusion, (c) Astro, (d) RadialExpansion.

4.1.4 Number of Blocks

With this factor, we consider the impact of the number of blocks on the inefficiency of the four algorithms. Five numbers of blocks are considered: 64, 128, 512, 1024, and 2048. For all five configurations, the total data size is 1024^3.

4.1.5 Number of Cells per Block

With this factor, we consider the impact of the block size (i.e., the number of cells per block) on the inefficiency of the four algorithms. Three sizes of blocks are considered: 64^3, 128^3, and 512^3. For each of these tests, we used 512 blocks.

4.1.6 Number of MPI Tasks

With this factor, we consider the scalability of our algorithms. This includes increasing the number of MPI tasks, as well as the data size and the number of particles. Three levels of concurrency are considered:

- Test 1: 32 MPI tasks, 5 lifelines, 1M particles, and 512 blocks.
- Test 2: 128 MPI tasks, 7 lifelines 4M particles, and 2048 blocks.
- Test 3: 512 MPI tasks, 9 lifelines 16M particles, and 8192 blocks.

4.2 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. There are 32 cores per node, and each core supports 2 hyper-threads and 128 GB of memory per node.

4.3 Algorithms Configuration

RSM-N and LSM request work from multiple victims at each request. LSM calculates the number of victims using the equation in Section 2.2. We used that equation to compute the lifeline graph with a base equals to 2. All our tests except for the last phase use 32 MPI ranks, and thus the number of lifelines is equal 5. For the RSM-N algorithm, we used the same number of victims as LSM (i.e., RSM-5).

4.4 Performance Measurement

For each phase, we measure the work time (including I/O, advection, etc.), idle time, and total time. From these measurements, we can derive the inefficiency of the algorithm. Inefficiency affects the performance because the execution time is determined by the time of the slowest processor. We define inefficiency with the following equation:

$$\text{Inefficiency} = \frac{\text{Idle}_{\text{time}}}{\text{Total}_{\text{time}}}$$

The idle time in our tests only measures the time that a node spends waiting for other nodes to finish their work. It does not include the time spent performing redundant I/O operations, which is more likely to happen with RSM, RSM-N, and LSM. Further, as the number of steal increases, it is more likely to perform redundant I/O. For this reason, we include in our results both total time and inefficiency. The goal of reducing inefficiency is to reduce the total execution time. It is important to make sure that the additional I/O and communication operations done to reduce inefficiency do not lead to a higher total execution time.

5 Results

In this section, we present the results of our study.

5.1 Phase 1: Base Case

Table 1: Comparing the performance of the four algorithms in terms of total execution time, the time for the individual routines, the idle time, and the inefficiency. The initialization time measures the time to initialize variables and generate initial seeds. The I/O time measures the time to read data blocks from disk or cache. The advection time measures the time to advect particles and to process the advection results (e.g., terminate). The communication time measures the time to request or send particles to other nodes and to inform other nodes of termination. The sorting time measures the time to sort particles by block after each round (line 18 in Pseudocode 1). The idle time measures the time where a node is waiting for other nodes to finish or send work. The inefficiency measures the percentage of execution time spent in idle and is computed as defined in Section 4.4.

	POP	RSM	RSM-5	LSM
Total time	131s	117s	111s	107s
Initialization time	1.68s	1.60s	1.55s	1.54s
IO time	14.8s	15.3s	22.9s	20.9s
Advection time	78.4s	75.0s	74.9s	73.6s
Communication time	$2.1e^{-4}$s	$7e^{-3}$s	0.15s	0.04s
Sorting time	9.21s	9.05s	8.80s	8.61s
Idle time	26.5s	16.3s	2.8s	1.7s
Inefficiency	0.20	0.14	0.03	0.02

In this phase, we compare the performance of the four algorithms, using the following configuration:

- Data set: Fishtank
- Number of particles: 1M
- Maximum advection steps: 10K
- Number of blocks: 512

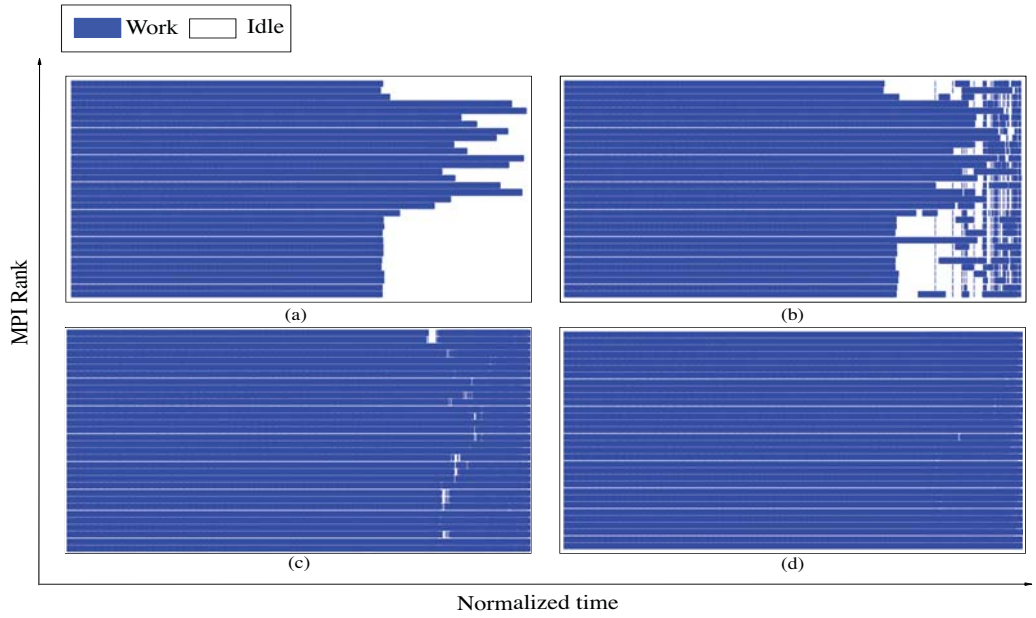

Figure 3: Performance of the four algorithms (a) POP, (b) RSM, (c) RSM-5, (d) LSM, using 32 MPI tasks to advect 1 million particles for 10 thousands steps (base case).

- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

The results of this phase are presented in Table 1. The results show a significant drop in inefficiency from 20% (POP) to 2% (LSM).

POP has the highest inefficiency, with 20% of the total execution time is spent idle. This is due to the load imbalance between nodes, which can be seen in Figure 3. Although the nodes have the same number of particles, their workload varies, see Section 3.2.1 for more discussion.

Using RSM reduces the inefficiency by a factor of 1.4. While this reduction improves the performance, idle time still takes 14% of the total execution time since thieves request work from one victim at a time.

Using multiple victims in RSM-5 reduces the inefficiency by a factor of 6.6 over POP, and a factor of 4.6 over RSM. This is because sending multiple requests at once allows the thief to receive work faster, and therefore reduces idle time.

LSM reduces the inefficiency by a factor of 10 over POP, a factor of 7 over RSM, and a factor 1.5 over RSM-5. LSM reduces the inefficiency compared to RSM-5 because the cyclic feature of the lifeline graph guarantees to always have a path from an idle node to a busy node.

Table 2: Comparing the number of advection steps and I/O operations between the four algorithms.

	POP	RSM	RSM-5	LSM
Total advection steps	9.97B	9.97B	9.97B	9.97B
Min advection steps	300M	231M	222M	230M
Max advection steps	312M	373M	383M	380M
Total # disk reads	3750	4081	5491	5381
Min # disk reads	67	87	126	128
Max # disk reads	256	221	216	198
Total # cache reads	2925	3009	3545	3587
Min # cache reads	1	4	30	28
Max # cache reads	244	223	195	207

I/O cost varies between the four algorithms. POP has the lowest I/O cost of all four algorithms, and it has the lowest number of I/O operations (disk and cache) as presented in Table 2. RSM has a higher I/O cost and I/O operations than POP. This increase is because particles are communicated between nodes and new data blocks are needed. RSM-5 and LSM have a higher I/O cost and I/O operations than RSM and POP. That is because more particles are communicated between nodes.

The advection time varies between the four algorithms, even though they are advecting the same number of particles. LSM does a better job balancing the workload, which leads to better usage of threads. On the other hand, when using POP the workload is not balanced, which leads to underused threads.

The communication cost varies between the three work requesting algorithms (RSM, RSM-5, LSM) because of the difference in their communication pattern. RSM has the lowest communication cost between the three algorithms since thieves communicate with one victim at a time. This results in a lower communication time at the cost of a higher idle time. Both RSM-5 and LSM communicate with the same number of victims at a single request. However, RSM-5 has a higher communication cost than LSM. This is because, in case of failure to receive work, LSM relies on its lifelines to receive work. RSM-5, on the other hand, needs to perform another request to 5 new victims until it receives work.

Even though the inefficiency is improving by a factor of 10, the total time is only improving by 20%. This is because there is a maximum improvement possible when improving a part of the program. This improvement is limited by the time needed to perform advection steps.

5.2 Phase 2: Data Sets

In this phase, we vary the data set using the following configuration:

- Number of particles: 1M
- Maximum advection steps: 10K
- Number of blocks: 512
- Number of cells per block: 128^3

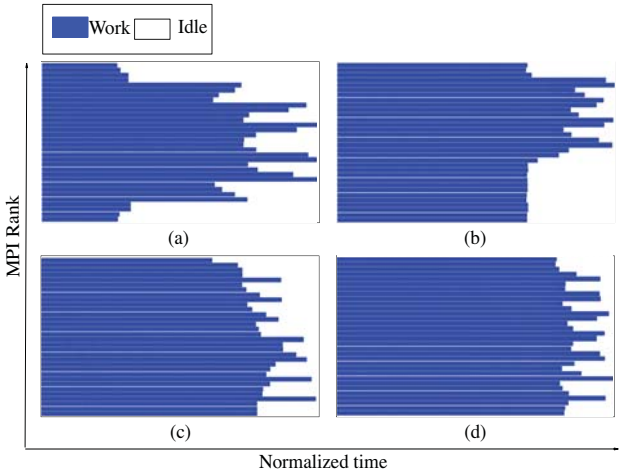

Figure 4: Performance of the POP algorithm on the four data sets (a) RadialExpansion, (b) Fishtank, (c) Astro, (d) Fusion, using 32 MPI tasks to advect 1 million particles for 10 thousands steps.

- Number of MPI tasks: 32 (512 cores)

Table 3: Comparing the inefficiency and time of the four algorithms when varying the data sets.

Data set		POP	RSM	RSM-5	LSM
Fishtank:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
Fusion:	Inefficiency	0.12	0.09	0.02	0.01
	Total time	115s	109s	102s	101s
Astro:	Inefficiency	0.18	0.12	0.03	0.02
	Total time	126s	120s	103s	102s
RadialExpansion:	Inefficiency	0.33	0.27	0.04	0.03
	Total time	74.1s	73.5s	60.8s	54.9s

The results of this phase are presented in Table 3. The table shows that LSM reduces the inefficiency for all four data sets and maintains low inefficiency ratios for all cases (0.01-0.03).

For the RadialExpansion data set, the POP algorithm has a high inefficiency ratio (30%). This is because the workload is highly imbalanced, as can be seen in Figure 4 (a). Since vectors are moving from the center toward the boundaries of the box, nodes that are responsible for particles located in the center of the box have a higher workload. RSM reduces the inefficiency by only a modest factor of 1.2 over POP. RSM-5 and LSM, however, reduce the inefficiency over POP by a factor of 8.2 and 11, respectively.

For the Fishtank data set, the POP algorithm also has a high inefficiency ratio (20%). This is because the workload is highly imbalanced, as can be seen in Figure 4 (b). The velocity field is moving toward a sink at one end of the box. Nodes that are responsible for particles that are located on the opposite side of the box have more workload. RSM reduces the inefficiency by a factor of 1.4 over POP. RSM-5 and LSM reduce the inefficiency over POP by a factor of 6.6 and 10, respectively.

For the Astro and Fusion data sets, the POP algorithm has a lower inefficiency ratio compared to the previous data sets: 18% for the Astro data set and 12% for Fusion data set. The workload of the POP algorithm is less imbalanced for these two data sets compared to the other two (Figure 4 (c) and (d)). This is because both data sets have a more uniform vector field. However, using RSM-5 and LSM still reduce the inefficiency significantly.

5.3 Phase 3: Number of Particles

In this phase, we vary the number of particles, using the following configuration:

- Data set: Fishtank
- Maximum advection steps: 10K
- Number of blocks: 512
- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

Table 4: Comparing the inefficiency and time of the four algorithms when varying the number of particles.

Number of Particles		POP	RSM	RSM-5	LSM
10K:	Inefficiency	0.25	0.23	0.17	0.11
	Total time	15.4s	14.2s	13.0s	12.1s
100K:	Inefficiency	0.28	0.20	0.15	0.07
	Total time	28.6s	26.6s	23.4s	22.6s
1M:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
10M:	Inefficiency	0.44	0.27	0.04	0.03
	Total time	1278s	921s	501s	486s

The results of this phase are presented in Table 4. The table shows that LSM reduces the inefficiency in all cases, with the highest improvement being a reduction of 14.6 over POP, for 10M particles.

The table shows that the inefficiency of POP is not directly correlated to the number of particles. At each test, the number of particles is increased by a factor of 10, but the inefficiency does not change within the same ratio and in one case it drops (in the case of 1M). This is because the inefficiency change is not dependent on the total number of particles, but rather on the workload distribution per node (Section 3.2.1). On the other hand, both RSM-5 and LSM are able to reduce the inefficiency consistently.

5.4 Phase 4: Number of Steps

In this phase, we vary the number of advection steps, using the following configuration:

- Data set: Fishtank
- Number of particles: 1M
- Number of blocks: 512
- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

Table 5: Comparing the inefficiency and time of the four algorithms when varying the durations of particles (maximum advection steps).

Advection Steps		POP	RSM	RSM-5	LSM
100:	Inefficiency	0.07	0.07	0.05	0.03
	Total time	20.3s	20.4s	20.2s	18.0s
1K:	Inefficiency	0.20	0.11	0.07	0.04
	Total time	33.7s	31.4s	29.3s	27.1s
10K:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
100K:	Inefficiency	0.19	0.12	0.02	0.01
	Total time	1080s	981s	877s	791s

The results of this phase are presented in Table 5. LSM reduces the inefficiency in all cases, with a reduction of a factor of 19 in the case of 100K steps.

For the case of 100 advection steps, POP has an inefficiency of 7%. This is because the number of advection steps is small, making it less likely for particles to travel across multiple blocks or exit the

domain. This reduces the probability of imbalance between nodes. RSM has the same inefficiency ratio as POP. This is because nodes have small workloads. Consequently, it is more difficult for a thief to find a victim with work available.

However, both RSM-5 and LSM reduce the inefficiency over POP by a factor of 1.4 and 2.3, respectively. As both methods are requesting work from five victims at a time, they are more likely to find work and therefore reduce the inefficiency.

Increasing the number of advection steps from 100 to 1k increases the inefficiency of POP to 20%. The table shows that the inefficiency of POP is not directly correlated to the number of advection steps but to the distribution of workload. On the other hand, both RSM-5 and LSM are able to consistently reduce the inefficiency down to 2% and 1%, respectively.

5.5 Phase 5: Number of Blocks

In this phase, we vary the number of blocks. That said, the overall data size remains constant through all tests (1024^3). The other factors for this configuration are the following:

- Data set: Fishtank
- Number of particles: 1M
- Maximum advection steps: 10K
- Number of MPI tasks: 32 (512 cores)

Table 6: Comparing the inefficiency and time of the four algorithms when varying the number of blocks.

Number of Blocks		POP	RSM	RSM-5	LSM
64:	Inefficiency	0.32	0.24	0.19	0.06
	Total time	89.1s	83.1s	82.7s	80.0s
128:	Inefficiency	0.25	0.21	0.09	0.04
	Total time	71.2s	68.9s	67.3s	65.1s
512:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
1024:	Inefficiency	0.24	0.16	0.02	0.01
	Total time	168s	151s	131s	122s
2048:	Inefficiency	0.30	0.16	0.01	0.01
	Total time	257s	219s	174s	172s

The results of this phase are presented in Table 6. It can be seen that LSM reduces inefficiency including by a factor of 30 for the largest number of blocks.

The results show that the inefficiency of the POP algorithm is not directly impacted by the change in the number of blocks. This is because, in the POP algorithm, nodes perform their computation independently without communicating with other nodes. Therefore loading more blocks on each node does not affect the overall load balance of the POP.

The inefficiency of the three other algorithms (RSM, RSM-5, LSM) reduces as the number of blocks increases. This is because these algorithms respond faster to work requests as the size of the block reduces. These algorithms run an iterative loop. At the end of each iteration, the nodes check for work requests and send the appropriate responses (Algorithm 2 line 17). Reading smaller blocks reduces the time spent in I/O, reducing the time between requests.

5.6 Phase 6: Cells per Block

In this phase, we fix the number of blocks to 512 and vary the size of blocks (i.e., data size), using the following configuration:

- Data set: Fishtank
- Number of particles: 1M
- Maximum advection steps: 10K
- Number of blocks: 512

Table 7: Comparing the inefficiency and time of the four algorithms when varying the number of cells per block with 512 blocks in total.

Cells per Block		POP	RSM	RSM-5	LSM
64^3:	Inefficiency	0.22	0.13	0.02	0.01
	Total time	131s	123s	100s	95.7s
128^3:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
256^3:	Inefficiency	0.36	0.23	0.05	0.04
	Total time	281s	249s	222s	212s

- Number of MPI tasks: 32 (512 cores)

The results of this phase are presented in Table 7. The table shows that LSM reduces the inefficiency for the different sizes of blocks, with a factor of 22 for the smallest size.

The inefficiency of the three other algorithms (RSM, RSM-5, LSM) increases as the size of blocks increases. When the size of the block increases, the time to read the block from disk increases. As described previously, spending more time in I/O increases the response time to work requests, leading to higher inefficiencies. RSM is the most impacted, because it sends only one request at a time, whereas RSM-5 and LSM are sending five requests at the same time. This increases the likelihood of RSM-5 and LSM to receive work faster.

5.7 Phase 7: MPI Tasks

In this phase, we vary the number of MPI tasks, as well as the number of particles and the data size, using the following configuration:

- Data set: Fishtank
- Maximum advection steps: 10K
- Number of cells per block: 128^3
  - Test 1: 32 MPI tasks (512 cores), 5 lifelines, 1M particles, and 512 blocks.
  - Test 2: 128 MPI tasks (2048 cores), 7 lifelines 4M particles, and 2048 blocks.
  - Test 3: 512 MPI tasks (8192 cores), 9 lifelines 16M particles, and 8192 blocks.

Table 8: Comparing the inefficiency and time of the four algorithms when varying the number MPI taks, number of particles and number of blocks.

# MPI tasks		POP	RSM	RSM-N	LSM
32:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
128:	Inefficiency	0.45	0.29	0.03	0.02
	Total time	315s	252s	209s	186s
512:	Inefficiency	0.54	0.36	0.06	0.05
	Total time	1439s	1012s	694s	671s

The results of this phase are presented in Table 8. The table shows that LSM reduces the inefficiency for the different test cases, with a factor of 10.8 for the largest test case.

The POP algorithm inefficiency increases as the test size increases, reaching 54% for the largest test case. As the size of the test increases, the difference in workload between the nodes increases, which can be seen in Table 9. This results in a higher load imbalance.

The POP algorithm is the most impacted by this imbalance. Using RSM reduces the inefficiency by a factor of 1.5 over POP in all cases. RSM still suffers from high inefficiencies (0.36 in the worst case). This is because the work becomes more sparse as the number of MPI

Table 9: Comparing the difference in workload balance between the four algorithms. The table shows the minimum work time, maximum work time, and the difference for each test. The work time in this table indicates the time spent in I/O and advection. The Diff measures the difference in time between the nodes with the highest and the lowest workloads.

# MPI tasks		POP	RSM	RSM-N	LSM
32:	Min work	82.7s	82.5s	90.1s	86.1s
	Max work	112s	101s	106s	100s
	Diff	29.3s	18.5s	15.9s	13.9s
128:	Min work	117s	112s	154s	139s
	Max work	241s	204s	186s	167s
	Diff	124s	92s	32s	28s
512:	Min work	451s	421s	525s	513s
	Max work	1209s	867s	598s	580s
	Diff	758s	446s	73s	67s

task increases. Consequently, thieves are less likely to randomly find a victim with work.

RSM-5 and LSM maintain low inefficiency ratios for all cases. Using RSM-5 reduces the inefficiency over POP by 6.6 for the first test, 15 for the second test, and 9 for the third test. Using LSM reduces the inefficiency over POP by 10 for the first test, 22.5 for the second test, and 10.8 for the third test. This is because both RSM-5 and LSM can find victims with work faster by requests work from multiple victims at a time.

5.8 Summary of Findings

The evaluation showed that the LSM algorithm reduces inefficiency in all cases. The algorithm adapts the number of its lifelines (victims) as the concurrency change to make sure there is a short path from busy nodes to idle ones. Further, in our largest test case (512 ranks, 16M particles, 17B cells data sets), LSM has the lowest inefficiency of all four algorithms.

Overall, the evaluation demonstrates that the LSM algorithm is a better choice for particle advection work requesting. LSM would be particularly well suited for production visualization tools. This is because the LSM algorithm can adapt itself to better support complex cases without requiring major user inputs. Further, production visualization tools must support a large variety of use cases, including those that lead to load imbalance with traditional approaches.

6 Conclusions and Future Work

The contribution of this paper is three-fold: (1) we designed a work requesting algorithm for parallel particle advection that uses lifeline-based scheduling (LSM) method, (2) we added an extension to random scheduling (RSM-N) to use multiple victims, and (3) we evaluated the efficiency of the three scheduling methods as well as POP. As discussed in the summary of findings, our LSM algorithm improves the performance compared to traditional approaches, especially on workloads that are prone to load imbalance.

For future work, we plan to implement a multi-threaded version of the algorithm with another thread for communication. Our tests in Phase 5 and 6 show that the LSM algorithm has lower inefficiency in cases where the algorithm performed smaller work at each iteration (reading smaller blocks), which reduces the response time to a work request. We plan to test the ideas suggested by Sisneros and Pugmire [22] where the algorithm advects a portion of the particles belonging to one block, to allow the algorithm to check for work requests more frequently. We also plan to study the impact of the number of liflines (victims) on the performance. Finally, we plan to test the algorithms at larger scale.

Acknowledgments

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] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999. doi: 10.1145/324133.324234

[2] 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 *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 57–64, Oct 2011. doi: 10.1109/LDAV.2011.6092318

[3] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy. Streamline integration using mpi-hybrid parallelism on a large multicore architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, Nov 2011. doi: 10.1109/TVCG.2010.259

[4] 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)*, pp. 1–8. Girona, Spain, May 2013.

[5] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *2008 IEEE Pacific Visualization Symposium*, pp. 87–94, March 2008. doi: 10.1109/PACIFICVIS.2008.4475463

[6] 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)*, pp. 1–10. Goa, India, Dec. 2014.

[7] E. Endeve, C. Y. Cardall, R. D. Budiardja, and A. Mezzacappa. Generation of Magnetic Fields By the Stationary Accretion Shock Instability. *The Astrophysical Journal*, 713(2):1219–1243, 2010.

[8] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale Algorithms for Reactor Hydrodynamics. *Journal of Physics: Conference Series*, 125:1–5, 2008.

[9] 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, pp. 10:1–10:11. ACM, New York, NY, USA, 2011. doi: 10.1145/2063384.2063397

[10] V. Kumar, K. Murthy, V. Sarkar, and Y. Zheng. Optimized distributed work-stealing. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, pp. 74–77. IEEE Press, Piscataway, NJ, USA, 2016.

[11] 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. doi: 10.1109/SC.2014.87

[12] H. Löffelmann, T. Kucera, and E. Gröller. Visualizing poincaré maps together with the underlying flow. In *In International Workshop on Visualization and Mathematics'97 Proceedings*, pp. 315–328. Springer, 1997.

[13] 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)*, 36(3):48–58, May/June 2016.

[14] C. Müller, D. Camp, B. Hentschel, and C. Garth. Distributed parallel particle advection using work requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 1–6, Oct 2013. doi: 10.1109/LDAV.2013.6675152

[15] 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*, 17(12):1785–1794, Dec 2011. doi: 10.1109/TVCG.2011.219

[16] 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. doi: 10.1109/IPDPS.2011.62

[17] 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. doi: 10.1145/1654059.1654076

[18] D. Pugmire, T. Peterka, and C. Garth. Parallel Integral Curves. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pp. 91–113. CRC Press/Francis–Taylor Group, Oct. 2012.

[19] D. Pugmire, A. Yenpure, M. Kim, J. Kress, R. Maynard, H. Childs, and B. Hentschel. Performance-Portable Particle Advection with VTK-m. In H. Childs and F. Cucchietti, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2018. doi: 10.2312/pgv.20181094

[20] A. Sanderson, G. Chen, X. Tricoche, D. Pugmire, S. Kruger, and J. Breslau. Analysis of recurrent patterns in toroidal magnetic fields. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1431–1440, 2010.

[21] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pp. 201–212. ACM, New York, NY, USA, 2011. doi: 10.1145/1941553.1941582

[22] R. Sisneros and D. Pugmire. Tuned to terrible: A study of parallel particle advection state of the practice. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1058–1067, May 2016. doi: 10.1109/IPDPSW.2016.173

[23] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, S. Plimpton, A. Tarditi, M. Chu, and the NIMROD Team. Nonlinear Magnetohydrodynamics with High-order Finite Elements. *J. Comp. Phys.*, 195:355, 2004.

[24] 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. doi: 10.1145/1362622.1362655

[25] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. A. Saraswat, and M. Takeuchi. Glb: lifeline-based global load balancing library in x10. In *PPAA@PPoPP*, 2014.