In Situ Particle Advection Via Parallelizing Over Particles

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

David Pugmire
pugmire@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Hank Childs
hank@uoregon.edu
University of Oregon
Eugene, Oregon

ABSTRACT

We extend the method for particle advection that parallelizes over particles to work in an in situ setting. We then compare our method with the typical method for in situ, parallelizing over data. Our experiments consist of parallelism at 512 cores, a data set with 67 million cells, and ten billion total advection steps. Our findings show that parallelizing over particles can be more than ten times faster for some workloads, for reasonable memory cost. Overall, the significance of these findings is to demonstrate that moving data can be worthwhile in some in situ settings.

CCS CONCEPTS

• **Human-centered computing → Scientific visualization**; • **Computing methodologies → Massively parallel algorithms**.

KEYWORDS

In situ visualization, parallel particle advection

ACM Reference Format:
Roba Binyahib, David Pugmire, and Hank Childs. 2019. In Situ Particle Advection Via Parallelizing Over Particles. In *ISAV: In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'19), November 18, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3364228.3364235

1 INTRODUCTION

Particle advection is a fundamental building block for many flow visualization algorithms. The term refers to displacing a massless particle according to a velocity field. In practice, this is accomplished by advancing a given particle in small steps, called advection steps. Advection steps calculate a particle's direction of travel by solving an ordinary differential equation using a numerical technique like Runge-Kutta. The resulting trajectory, or trajectories when there is a set of particles, can then be used as the basis for flow visualization algorithms, such as streamlines, pathlines, stream surfaces, FTLE, etc.

Some particle advection workloads require many advection steps, and thus can be very computationally expensive. Some flow visualization algorithms require millions, or possibly even billions, of particles. Other algorithms advect the particles for long distances, requiring tens of thousands of steps per particle. In some cases, the technique requires both many particles and long distances. Worse, computational meshes can be very fine, containing billions of cells; these meshes often are too large to fit into memory, and thus need to be decomposed into blocks. Further, particles will move from block to block with regularity, which creates difficulties in designing algorithms. For particle advection problems with many advection steps and fine computational meshes, the most common solution is to incorporate parallel processing.

Most of the work over the last two decades on parallelizing particle advection algorithms has come in the context of post hoc processing. In the post hoc setting, there is typically enough available memory for a given processing element (i.e., MPI task) to load multiple blocks, and also to store blocks redundantly across the processing elements. Further, acquiring a given block in a post hoc setting typically means reading it from disk, meaning that all blocks acquisitions (reads) take the same amount of time.

The assumptions made by post hoc algorithms change in a "tightly-coupled" in situ setting (i.e., where the simulation code and visualization routines share the same memory space). First, memory is assumed to be very precious because it is shared with the simulation code, which discourages acquiring multiple blocks and also having redundant blocks. Second, each processing element already has one block (i.e., the one the simulation code is operating on) and so the assumption is that the visualization routines should also operate on that same block, to save on memory. Finally, block acquisitions would no longer translate to reading data from disk, but instead acquiring data from another processing element via network communication.

Only one of the existing particle advection parallelization methods, parallelize-over-data (POD), aligns with in situ constraints. With this method, each processing element operates on a given block (or blocks) and particles are sent over the network as they move between blocks. In the tightly-coupled in situ setting, the block for a given processing element would be the same one the simulation code is operating on, minimizing memory usage.

The purpose of this work is to explore whether choices aside from POD are suitable for tightly-coupled in situ processing as well. While POD will minimize memory usage, it may be a poor choice with respect to execution time, which is also a very important consideration. In particular, POD performs poorly when particles are located in a small subset of the blocks, as this condition creates load imbalance.

To explore this theme, we introduce a straightforward variant of the parallelize-over-particle (POP) algorithm that is appropriate for in situ processing. The basic idea of POP is that particles are distributed among processing elements, and each processing element advects its particles, fetching data blocks as needed to resolve the velocity field surrounding its particles. The key difference between our in situ algorithm and traditional (post hoc) POP is that in our algorithm a block is acquired via network communication from another processing element, while traditional POP acquires blocks from disk. In our experiments, we allowed each processing element to store up to two additional blocks (costing 40MB each), and found that runtimes improved by 10X over POD for some workloads. While this additional memory overhead may be prohibitive in some settings, we feel our approach is useful in the settings where there is available memory.

Overall, we feel the main contribution of this work is to show that the wide body of previous research on parallelizing particle advection from the post hoc setting may still have a place in an in situ setting.

2 RELATED WORK

2.1 Parallel Particle Advection

There are only two main parallelization approaches for particle advection (POD and POP), although these approaches have multiple variants [15]. Most extensions to these two approaches aim to improve load balance. For POD, improvements have come via techniques such as round-robin block assignment [13] and preprocessing [4, 12, 21]. For POP, extensions have included techniques to balance the workload, such as work requesting [8, 11], and dynamic load balancing [10]. Other extensions focused on reducing I/O cost by using techniques such as extending the memory hierarchy [3], and data prefetching [2, 17]. Finally, some studies proposed hybrid algorithms [6, 14], where the algorithm used both parallelization techniques (POP and POD) to balance the workload.

2.2 In Situ Parallel Particle Advection

Several works have employed particle advection techniques in situ. Most notably, Vetter et al. [20] presented an in situ framework for large unsteady flow data. Their solution used POD as a parallelization method. Further, an emerging in situ data reduction approach for vector fields uses parallel particle advection to calculate Lagrangian basis flows [1, 18, 19]. These works also use POD.

3 ALGORITHM

In this section, we present our POP implementation for an in situ context. For ease of reference, we abbreviate the term Processing Element as "PE" in our description. A PE equates to one MPI task. It also could equate to one compute node, provided there is one MPI task per node.

As discussed earlier, POP distributes particles across PEs, and the needed data blocks are acquired by each PE on demand. In a post hoc context, the data block is acquired by reading data from disk. To adapt the algorithm to work in an in situ framework, PEs in our algorithm acquire needed data blocks from other PEs. We hypothesize that improvements in load balance will offset the cost of communicating data blocks, which can be high. Finally, our algorithm dedicates a separate thread for communication to hide the communication cost.

An important consideration for in situ POP is memory consumption. A PE acquiring many data blocks runs the risk of exceeding the budget allocated by the simulation for in situ processing. Instead, total memory needs to be controlled. Our algorithm allows the user to set the number of data blocks allowed in memory of a given PE. Before each data request, the algorithm checks if there is available space to make sure not to exceed the number of allowed data blocks. If the algorithm reached the maximum number of data blocks, it removes a block to make space for the new block. For our experiments, we set the maximum block size at two.

Algorithm 1 shows the pseudocode for the worker thread. It uses the following building blocks:

- Particle: a data structure that represents a particle in the vector field. The structure contains the particle id, position, current block id, and can also store the trajectory of the particle.
- ParticleArray: a data structure that stores an array of Particles.
- ArrayOfParticleArrays: a data structure that stores multiple elements of ParticleArray. Each of these elements stores multiple elements of Particle.
- SortParticleByBlock(): a function that sorts Particles depending on their current block id and returns two elements: ArrayOfParticleArrays and a vector containing the ids of needed blocks. All Particles that belongs to block i are stored in index i of ArrayOfParticleArrays.
- Advect(): a function that advects the Particles of a ParticleArray until they exit the current block or terminate. This function returns two ParticleArray elements: the first one contains the completed particles, and the second one contains particles that need another data block.
- CheckForIncomingMessages(): a function that checks for incoming messages from other PEs. These messages can be data requests from other PEs or notifications of particle terminations.
- SendData(): a function that sends a data block to the requesting PE.
- RequestData(): a function that requests a data block from another PE.

The algorithm starts by distributing P particles across N PEs, assigning $\frac{P}{N}$ particles to each PE. Each PE then begins the process of advecting its particles. First, each PE starts by sorting particles by block and identifying the needed data blocks. Next, the worker thread advects the particles located in its local data block. We use the VTK-m [9] library for particle advection within a PE, specifically the module developed by Pugmire, et al. [16]. Simultaneous to advection, the communication thread requests needed data blocks; this is described in Algorithm 2. When a PE receives a requested data block, the PE's particles located in that data block would be advected. The algorithm completes when all particles are terminated, either by reaching the maximum advection step or exiting the problem domain.

An important consideration for our algorithm was the cost to send data. When a PE's block is requested, it employs a multi-threaded approach to serialize the data into a byte string. It also caches this byte string to prevent repeated serialization costs.

4 EXPERIMENTAL OVERVIEW

This section provides an overview of our experiments: experiment configurations (4.1) and the metrics we use to evaluate performance (4.2).

Algorithm 1 Pseudocode of the worker thread for one PE.

```
function POP-Advect(ParticleArray pv)
    $keepGoing \leftarrow true$
    $ArrayOfParticleArrays\ \ pva[NUMBLOCKS]$
    $(pva, neededDataBlocks) \leftarrow SortParticlesByBlock(pv)$
    $allCompletedParticles \leftarrow \emptyset$
    while keepGoing do
        $contParticles \leftarrow \emptyset$
        for i in NUMBLOCKS do
            if pva[i].size() > 0 then
                $ParticleArray\ \ completed,\ \ continuing$
                $(completed, continuing) \leftarrow Advect(pva[i], b)$
                $allCompletedParticles\ \ += \ \ completed$
                $contParticles\ \ += \ \ continuing$
            end if
        end for
        if contParticles.size() > 0 then
            $pva \leftarrow SortParticlesByBlock(contParticles)$
        else
            $keepGoing \leftarrow false$
        end if
    end while
end function
```

Algorithm 2 Pseudocode of the communication thread for one PE.

```
function POP-Communicate(int* neededDataBlocks)
    for i in neededDataBlocks do
        $owner \leftarrow GetOwnerNode(i)$
        $dataBuffer \leftarrow RequestData(owner, i)$
    end for
    if numActive > 0 then
        $keepCommunicating \leftarrow true$
    end if
    while keepCommunicating do
        $MSG \leftarrow CheckForIncomingMessages()$
        if MSG = PARTICLES_TERMINATED then
            numActive -= MSG.numTerminated
        else if MSG = NEED_DATA then
            $SendData(MSG.blockID)$
        end if
        if numActive < 0 then
            $keepCommunicating \leftarrow false$
        end if
    end while
end function
```

4.1 Experiment Configurations

Data Set: Our study used an astrophysics data set consisting of 32 blocks, with each block containing 128^3 cells. It came from a simulation data of a magnetic field surrounding a solar core collapse, which results in a supernova. The simulation was performed via the GenASiS [5] code, which is a multi-physics code for astrophysical systems involving nuclear matter.

Level of concurrency: We ran all experiments using 32 MPI tasks on 16 nodes of Cori, a machine at Lawrence Berkeley National Laboratory's NERSC facility. Cori has both Xeon Phi and Intel Xeon "Haswell" processor nodes; our experiments were run on the Haswells. We used 16 cores per MPI task, for a total of 512 cores in each run. We declined to use the hyper-threading feature, since it did not boost performance for the VTK-m code base we were using. Each Haswell node on Cori has 128GB of memory.

Parallelization Techniques: We consider both the POD algorithm and the POP extension we introduced in this paper. The POP algorithm running on each PE was allowed to cache up to two blocks it acquired from other PEs. While in this study we limited the cache size to two additional blocks, the user can choose to increase the number of allowed data blocks in cache to improve performance but at the cost of a higher memory consumption. It is important to note that while our POP algorithm is designed for in situ, we ran in a so-called "theoretical" in situ environment, as our algorithm was not connected to a running simulation. Instead, before executing the algorithm, each PE acquired one block of data from disk. From this point forward, the disk was not consulted, and data was exchanged via network as it would be in an in situ setting. No I/O timings are reported, since we feel it is not relevant to our study.

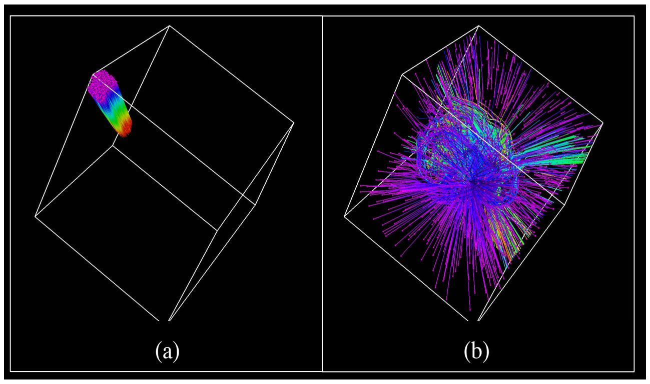

Figure 1: Streamlines visualization for our (a) dense and (b) uniform seed distributions.

Particle Workload: We used one million particles, and advected each particle 10K steps (or fewer in the relatively rare cases where a particle exited the volume), for a total of approximately 10 billion advection steps. Particles were advected using velocity. We consider two extremes of seeding distributions: dense and uniform. In our study, the dense distribution was so concentrated that all of the particles begin in a single block, which is very likely to lead to load imbalance when using POD. Our uniform distribution had particles spread evenly throughout the volume, increasing communication cost when using POP, since more data blocks are required. Figure 1 shows a visualization of the two distributions.

4.2 Performance Measurement

For each phase, we display the execution time of the slowest PE, the maximum memory consumption needed to store the data, and the load imbalance. The load imbalance impacts the performance because the execution time is determined by the time of the slowest

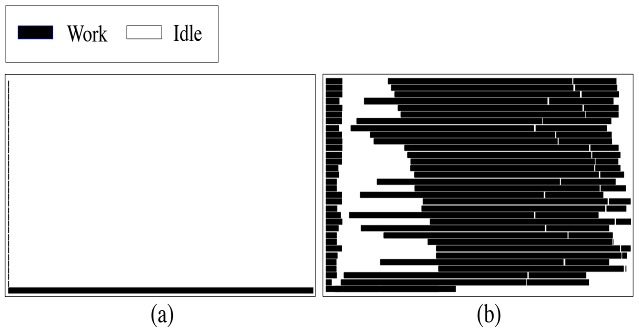

Figure 2: Performance of the two algorithms (a) POD, (b) POP, using 32 PEs to advect 1 million particles for 10 thousands steps for a dense distribution of seeds. The POD figure shows one task working the whole time (the task at the bottom), while the POP figure has more PEs involved. This figure is horizontally scaled based on run-time; POD ran for 307s, while POP ran for 26.6s.

PE. We define load imbalance with the following equation:

$$\text{Load imbalance} = \frac{\mathrm{T_s}}{\sum_{0<p<N} \mathrm{T_p}/\mathrm{N}}$$

where T_p is the total execution time for PE P, and T_s is the total execution time of the slowest PE.

5 RESULTS

This section presents the results of our study. We divide our analysis based on the seed distribution: dense (5.1) and uniform (5.2).

5.1 Dense Distribution

Table 1: Comparing the performance and memory consumption of the two algorithms for a dense particle distribution. Initialization time measures the time to initialize variables and generate initial seeds. Advection time measures the time to advect particles and to process the advection results (e.g., terminate). Communication time measures the time to request or send data blocks or particles to other PEs and to inform other PEs of termination. Idle time is the difference between total time and the sum of the other time measurements.

	POD	POP
Total time	307s	26.6s
Initialization time	0.97s	0.33s
Advection time	303s	16.9s
Communication time	0.18s	4.22s
Sort particle time	0.02s	0.1s
Load imbalance	30.34x	1.2x
Memory to store data	46.99MB	93.99 MB

The results for dense seeding are presented in Table 1. The results show that using POP improves performance by a factor of 11.5X over POD.

POD has a high execution time of 307s, due to the high load imbalance between PEs. This phenomenon is plotted in Figure 2. Since all particles are located in one data block (*block*0), there is one PE advecting all particles.

Using POP distributes the workload and reduces the execution time to 26.6s. Even though the communication takes 4.2s, the overall execution time is lower than POD. As discussed in Section 3, we took care to optimize serialization time, which is an important component of communication time. We found that serializing a 128^3 data block took about one-eighth of a second. (Previous versions over our code that serialized with a single core were much slower.) Figure 2 shows that there is idle time for each PE after advecting the particles located in its block. This idle time is the time spent waiting to receive the required data block.

Using POP increases the memory requirement needed to store the data. This is because each PE is storing its data block and its received data blocks. In the case of dense distribution, only one extra block was needed, meaning that the cache of size two was only half-filled. The memory consumption presented in the table is representing the number of MB needed to store the velocity data; if a simulation code was calculating extra quantities (temperature, density, etc.), then the proportional increase in memory would be lower.

5.2 Uniform Distribution

Table 2: Comparing the performance and memory consumption of the two algorithms for uniform seeding. The terms in this table are described in Table 1.

	POD	POP
Total time	23.9s	210s
Initialization time	0.65s	0.75s
Advection time	17.6s	99.7s
Communication time	4.34s	21.8s
Sort particle time	0.01s	13.9s
Load imbalance	1.19x	1.41x
Memory to store data	47.12MB	140.9MB

The results for uniform seeding are presented in Table 2. With uniform seeding, POD performs 9.9X better than POP. This is because POP's PEs needed to request data blocks from the other 31 PEs, since its particles are scattered across the whole data domain. This leads to a higher communication time, in addition to idle time waiting for data blocks.

In this test, the PEs made use of both slots in its cache. This means at any given time, each PE could store a maximum of 140.9MB of vector field data. We anticipate that a larger cache could substantially reduce execution time. When the size is small, a smaller number of blocks can be requested at the same time, since our algorithm checks for available slots before each request. As a result, PEs might need to request the same data block more than once for cases where particles advect toward a previous data block.

6 CONCLUSION AND FUTURE WORK

The contribution of this paper is an extension of the POP algorithm to work on an in situ context. We adapt the algorithm to acquire data blocks from other PEs instead of reading it from disk. The paper compares between the main particle advection parallelization methods (POD and POP), and shows that our POP extension is superior for the workload where POD is known to perform poorly. Further, the study provides evidence that other parallelization techniques designed for post hoc processing may also be useful for in situ processing.

For future work, we plan to integrate our algorithm with Ascent [7] and study the performance with more simulations. We also plan to study the impact of cache size (i.e., memory usage) on the performance. Finally, we plan to test the algorithms at larger scale.

7 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] Alexy Agranovsky, David Camp, Christoph Garth, E. Wes Bethel, Kenneth I. Joy, and Hank Childs. 2014. Improved Post Hoc Flow Analysis via Lagrangian Representations. In *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV).* 67–75.

[2] O. O. Akande and P. J. Rhodes. 2013. Iteration aware prefetching for unstructured grids. In *2013 IEEE International Conference on Big Data.* 219–227. https://doi.org/10.1109/BigData.2013.6691578

[3] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. I. Joy. 2011. Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *2011 IEEE Symposium on Large Data Analysis and Visualization.* 57–64. https://doi.org/10.1109/LDAV.2011.6092318

[4] Li Chen and I. Fujishiro. 2008. Optimizing Parallel Performance of Streamline Visualization for Large Distributed Flow Datasets. In *2008 IEEE Pacific Visualization Symposium.* 87–94. https://doi.org/10.1109/PACIFICVIS.2008.4475463

[5] Eirik Endeve, Christian Y. Cardall, Reuben D. Budiardja, and Anthony Mezzacappa. 2010. Generation of Magnetic Fields By the Stationary Accretion Shock Instability. *The Astrophysical Journal* 713, 2 (2010), 1219–1243.

[6] Wesley Kendall, Jingyuan Wang, Melissa Allen, Tom Peterka, Jian Huang, and David Erickson. 2011. Simplified Parallel Domain Traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11).* ACM, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/2063384.2063397

[7] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV'17).* ACM, New York, NY, USA, 42–46. https://doi.org/10.1145/3144769.3144778

[8] K. Lu, H. Shen, and T. Peterka. 2014. 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.* 1008–1019. https://doi.org/10.1109/SC.2014.87

[9] Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. 2016. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)* 36, 3 (May/June 2016), 48–58.

[10] D. Morozov and T. Peterka. 2016. Efficient Delaunay Tessellation through K-D Tree Decomposition. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 728–738. https://doi.org/10.1109/SC.2016.61

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

[12] B. Nouanesengsy, T. Y. Lee, and H. W. Shen. 2011. Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec 2011), 1785–1794.

[13] T. Peterka, R. Ross, B. Nouanesengsy, T. Y. Lee, H. W. Shen, W. Kendall, and J. Huang. 2011. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *2011 IEEE International Parallel Distributed Processing Symposium.* 580–591. https://doi.org/10.1109/IPDPS.2011.62

[14] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. 2009. Scalable computation of streamlines on very large datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* 1–12. https://doi.org/10.1145/1654059.1654076

[15] David Pugmire, Tom Peterka, and Chistoph Garth. 2012. Parallel Integral Curves. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight.* CRC Press/Francis–Taylor Group, 91–113.

[16] David Pugmire, Abhishek Yenpure, Mark Kim, James Kress, Robert Maynard, Hank Childs, and Bernd Hentschel. 2018. Performance-Portable Particle Advection with VTK-m. In *Eurographics Symposium on Parallel Graphics and Visualization*, Hank Childs and Fernando Cucchietti (Eds.). The Eurographics Association. https://doi.org/10.2312/pgv.20181094

[17] Philip J. Rhodes, Xuan Tang, R. Daniel Bergeron, and Ted M. Sparr. 2005. Iteration Aware Prefetching for Large Multidimensional Datasets. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM'2005).* Lawrence Berkeley Laboratory, Berkeley, CA, US, 45–54. http://dl.acm.org/citation.cfm?id=1116877.1116883

[18] Sudhanshu Sane, Roxana Bujack, and Hank Childs. 2018. Revisiting the Evaluation of In Situ Lagrangian Analysis. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV).* Brno, Czech Republic, 63–67.

[19] Sudhanshu Sane, Hank Childs, and Roxana Bujack. 2019. An Interpolation Scheme for VDVP Lagrangian Basis Flows. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV).* Porto, Portugal, 109–118.

[20] Vetter and Olbrich. 2017. Development and Integration of an In-Situ Framework for Flow Visualization of Large-Scale, Unsteady Phenomena in ICON. *Supercomput. Front. Innov.: Int. J.* 4, 3 (Sept. 2017), 55–67. https://doi.org/10.14529/jsfi170303

[21] H. Yu, C. Wang, and K. Ma. 2007. Parallel hierarchical visualization of large time-varying 3D vector fields. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing.* 1–12. https://doi.org/10.1145/1362622.1362655