Performance-Portable Particle Advection with VTK-m

David Pugmire[1], Abhishek Yenpure[2], Mark Kim[1], James Kress[1,2], Robert Maynard[3], Hank Childs[2] and Bernd Hentschel[4]

[1]Oak Ridge National Laboratory, Oak Ridge TN, USA
[2]University of Oregon, Eugene OR, USA
[3]Kitware, Inc. Clifton Park, NY, USA
[4]RWTH Aachen University, Aachen, Germany

Abstract

Particle advection is the fundamental kernel behind most vector field visualization methods. Yet, the efficient parallel computation of large amounts of particle traces remains challenging. This is exacerbated by the variety of hardware trends in today's HPC arena, including increasing core counts in classical CPUs, many-core designs such as the Intel Xeon Phi, and massively parallel GPUs. The dedicated optimization of a particle advection kernel for each individual target architecture is both time-consuming and error prone. In this paper, we propose a performance-portable algorithm for particle advection. Our algorithm is based on the recently introduced VTK-m system and chiefly relies on its device adapter abstraction. We demonstrate the general portability of our implementation across a wide variety of hardware. Finally, our evaluation shows that our hardware-agnostic algorithm has comparable performance to hardware-specific algorithms.

1. Introduction

In order to keep up with the amounts of raw data generated by state of the art simulations, modern visualization algorithms have to be able to efficiently leverage the same, massively parallel hardware that is used for data generation, i.e. today's largest supercomputers. This holds true for both classical *post processing* and modern *in situ* strategies. Specifically, as the latter have to run *at simulation time*, they have to be able to deal with a large variety of hardware platforms efficiently. Even more challenging, as visualization is oftentimes not seen as a first class citizen, it might have to run on different resources whenever they are available, e.g. utilizing idle CPU cores while the simulation is advanced on GPUs or using a local GPU while the simulation is blocked by communication. Custom-tailoring visualization algorithms to specific hardware platforms and potential usage scenarios is both time-consuming and error-prone. This leads to a gap with regard to practically available visualization methods for large data: either they are portable across a number of architectures, but do not feature ultimate performance, or they are highly-optimized, yet work only on a very limited subset of today's diverse hardware architectures. Performance-portable formulations of key algorithms have the potential to bridge this gap: the developer specifies *what* can be run in parallel while an underlying run-time system decides the *how* and *where*. Further, the runtime system is optimized *once* to make good use of a specific target architecture. Ideally, the result is that all previously formulated kernels will be available — with good, if not fully optimal, performance — on the newly addressed system. In the past, this approach has been demonstrated to show good results for inherently data parallel visualization problems, e.g. ray tracing [LMNC15] and direct volume rendering [LLN*15].

In this paper, we extend the body of performance-portable approaches with a method for parallel particle advection. Particle advection is the basic algorithmic kernel of many vector field visualization techniques. Applications encompass, e.g., the direct representation of field lines [SKH*05, EBRI09], dense vector field visualization methods [CL93, vW02, LEG*08], flow surfaces [FBTW10, GKT*08, Hul92], and the computation of derived data fields or representations [ACG*14, GGTH07, Hal01, SP07] or statistical measures [WP06, Wan10]. These techniques depend on the ability to compute large numbers of particle trajectories through a vector field. The resulting workloads are taxing with respect to both their computational requirements and their inherent dependence on high data bandwidth. In contrast to the aforementioned visualization kernels — isosurfacing and ray casting — particle tracing computations are not trivially data parallel. Worse, workloads are highly dynamic, as the outcome inherently depends on the input vector field. The computations' overall demands are therefore hard to predict in advance. This complicates — among other things — the efficient scheduling of parallel particle advection computations. In the recent past, a variety parallelization strategies has been proposed, all sharing the goal of providing many particles fast (see, e.g., [BSK*07, PCG*09, CGC*11, CKP*13, CSH16, NLS11, NLL*12, MCHG13]).

Our approach is based on the *parallelize-over-seeds* (POS) strategy. In order to facilitate performance portability, we rely on the concepts of the recently introduced VTK-m framework [MSU*16].

This facilitates the direct use of classical CPUs and GPUs without the need to implement and maintain two distinct code paths. We evaluate our algorithm's performance for a set of five well-defined workloads which cover a wide range of vector field visualization tasks. Our results show that our technique performs as expected, and matches the performance characteristics identified in previously published work. Further, we demonstrate that there are minimal performance penalties in our portable implementation when compared to hand-coded reference implementations.

To summarize, we make the following contributions. We propose a performance-portable formulation of POS particle tracing embedded in the VTK-m framework. We demonstrate general effectiveness and performance-portability based on the results of several performance experiments. As part of these experiments, we assess the cost of performance portability by comparing our method's performance to that of native implementations on a variety of execution platforms. Against this backdrop, we discuss the advantages and limitations of our approach with respect to natively-optimized techniques.

2. Related Work

In this section, we briefly review related work under two different aspects: parallel visualization systems and particle advection.

Parallel Visualization Systems The increasing need for production-ready, scalable visualization methods led to the development of several general purpose packages such as Paraview [ABM*01, Aya15] and VisIt [CBB*05]. These tools have primarily focused on distributed-memory parallelism, which is complementary to our own focus.

Recent changes in both HPC hardware and software environments led to the development of new frameworks, which address certain aspects of the changing HPC environment.

The DAX toolkit [MAGM11], introduced by Moreland et al., is built around the notion of a *worklet*: a small, stateless construct which operates — in serial — on a small piece of data. Worklets are run in an execution environment under the control of an executive. They help programmers to exhibit fine-grained data parallelism, which is subsequently used for data-parallel execution.

Lo et al. proposed to use data parallel primitives (DPP) [Ble90] for a performance-portable formulation of visualization kernels [LSA12] in the Piston framework. Based on NVidia's Thrust library [BH11], Piston supports both GPUs and CPUs as target architectures, by providing a CUDA and an OpenMP backend, respectively. Performance results demonstrated the ability to achieve good performance on different platforms using the exact same source code for each.

Meredith et al. introduce EAVL, a data parallel execution library with a flexible data model that addresses a wide range of potential data representations [MAPS12]. With this data model, they aim at increased efficiency – both in terms of memory use and computational demand – and scalability. The generic model proposed for EAVL allows developers to represent (almost) arbitrary input data in a way that accounts for hardware-specific preferences. For example, it allows them to switch between *structure-of-arrays* and *array-of-structures* representations of multi-dimensional data fields. Low-level parallelism is supported by an iterator-functor model: iteration happens in parallel, applying a functor to each item of a range.

Eventually, the experiences gathered in the development of these libraries resulted in the consolidated development of VTK-m [MSU*16]. It integrates an evolution of EAVL's data model with the two-tier control/execution environment of DAX and the idea to facilitate performance portability by formulating visualization workloads in terms of data parallel primitives. Currently, it offers parallel backends for CUDA and Intel Threading Building Blocks (TBB). Moreland et al. discuss the cost of portability for a variety of visualization workloads. Their findings are inline with and partially based on work by Larsen et al., who studied DPP-based formulations for ray tracing and direct volume rendering, respectively [LMNC15, LLN*15]. In this paper, we focus on an efficient formulation of a basic, general-purpose particle advection kernel running in a shared memory parallel environment. Hence, we chose VTK-m as our development platform.

Particle Advection The computation of integral lines is a fundamental kernel of vector field visualization [MLP*10]. Pugmire et al. review the two fundamental approaches – *parallelize-over-seeds* (POS) and *parallelize-over-blocks* POB – in a distributed memory environment and introduce a hybrid master-slave scheme that addresses load-balancing issues [PCG*09]. POS distributes the seeds of a target particle population across processing elements (PEs) and computes them independently of each other. In contrast, POB assigns the individual blocks of a domain decomposition to PEs; then, each PE is responsible for generating the traces that enter one of its assigned blocks. The newly introduced hybrid scheme, which dynamically – and potentially redundantly – assigns blocks to PEs addresses load-balancing problems that typically become a problem for pure POB while also limiting redundant I/O operations which oftentimes limit POS' scaling. An overview of distributed memory parallel particle advection methods is given in [PPG12].

Camp et al. propose a two-tier scheduling scheme: they use MPI to parallelize computations across multiple nodes and then execute local advection using OpenMP-parallel loop constructs [CGC*11]. Subsequently, Camp et al. analyze the effects exchanging the OpenMP-based advection for a CUDA-based solution [CKP*13]. We use a refined version of Camp's original CUDA advection scheme for comparisons in Sec. 4.

Kendall et al. propose a method inspired by MapReduce to parallize the domain travseral inherent to parallel particle tracing [KWA*11].

Yu et al. propose a data-dependent clustering of vector fields which enables the development of a data-aware POB strategy [YWM07]. Nouanesengsy et al. accumulate information about the probable propagation of particles across blocks and subsequently formulate the task of partitioning the set of blocks to PEs as an optimization problem [NLS11]. Subsequently, Nouanesengsy et al. propose to extend the basic POB idea to the time dimensions, distributing time intervals to PEs in order to generate the pathlines needed to compute the Finite Time Lyapunov Exponent (FTLE) [NLL*12].

Guo et al. propose a method using K-D tree decompositions to

dynamically balance the workload for both steady and unsteady state particle advection [ZGY*17].

Mueller et al. investigate the use of an alternative, decentralized scheduling scheme, *work requesting*: whenever a process runs out of work – i.e. particles to integrate – it randomly picks a peer and requests half its work in order to continue [MCHG13]. Hence scheduling overhead only occurs when there is actual imbalance in the system.

Being a key computational kernel for an array of vector field visualization algorithms, the optimization of particle advection for different architectures has garnered significant interest in the visualization research community. Initially, this was fueled by the advent of modern, programmable graphics hardware (GPUs). Interactive particle integration has been targeted for steady-state uniform grids [KKKW05], time-varying uniform grids [BSK*07], and tetrahedral meshes [SBK06], respectively. Bussler et al. propose a CUDA formulation for particle advection on unstructured meshes and additionally investigate the use of 3D mesh decimation algorithms in order to reduce the GPU memory requirements. Hentschel et al. propose the tracing of particle packets which facilitates the use of SIMD extensions in modern CPU designs [HGK*15]. They found that significant gains can be achieved specifically due to the optimization of memory accesses. Chen et al. follow a similar idea: they propose to integrate spatially coherent bundles of particles through time-varying, unstructured meshes in order increase memory locality on the GPU [CSH16].

The studies presented in [CKP*13, CBP*14] compare CPU and GPU-based hardware architectures w.r.t. their suitability for parallel integral curve computations. Sisneros et al. performed a parameter space study for a round-based POB advection algorithm [SP16]. Their findings suggest that naïvely chosen default settings – e.g. advecting all particles in each round – often lead to significantly degraded performance.

In summary, we find a great variety of optimization efforts targeting the important yet seemingly inconspicuous computational kernel of particle integration. In light of studies like [CKP*13, CBP*14, HGK*15], we argue that making good use of any new hardware architecture or even of new features on the one hand requires an intimate knowledge of said features and on the other hand can be very time consuming, particularly due to the required low-level programming. This observation provides the major motivation for the performance-portable approach proposed in the following section.

3. Parallel Particle Advection in VTK-m

As stated above, the VTK-m framework is a response to the growing on-node parallelism that is available on a wide variety of new architectures. In order to be able to efficiently cater to a variety of different hardware platforms, VTK-m relies on the concept of data parallel primitives. VTK-m distinguishes two different realms: the control environment and the execution environment. The control environment is the application-facing side of VTK-m. It contains the data model and allows application programmers to interface with algorithms at large. In contrast, the execution environment contains the computational portion of VTK-m. It is designed for massive parallelism. Worklets (c.f. Sec. 2) are an essential part of

```
INPUTS:
Seed, VectorField, NumberOfSteps, StepSize
OUTPUTS:
Advected

Advected = Advect(Seed, VectorField,
                  NumberOfSteps, StepSize)

Function:
Advect(pos, vectorField, numberOfSteps, stepSize):
  S = 0
  while S < numberOfSteps :
  {
    if pos in vectorField :
      newPos = RK4(pos, stepSize, vectorField)
      pos = newPos
      S = S+1
    else :
      break
  }

  return pos

Function:
RK4(p, h, f)
    k1 = f(p)
    k2 = f(p + (h/2)k1)
    k3 = f(p + (h/2)k2)
    k4 = f(p + hk3)
    return (h/6)(k1 + 2k2 + 2k3 + k4)
```

Figure 1: *Pseudo code for our implementation of* `Advect`*, our particle advection for a routine parallelize-over-seed (POS) strategy. This routine operates on a single particle and provides the definition of our elementary unit of work.*

the execution environment, and are the mechanism for performing operations on elements of the data model in parallel. Finally, device adapters provide platform-specific implementations of generic DPPs and memory management. Specifically — where necessary — they abstract the transfer of data between host and device memory.

Algorithms in VTK-m are created by specifying a sequence of DPP operations on data representations. When compiled, the parallel primitives are mapped to the particular device implementation for each primitive. This indirection limits the amount of optimization that can be done for a particular algorithm on a particular device, which in turn raises the question of costs for performance portability. In the following, we describe a DPP-based realization of parallel particle advection which will eventually lead to a parallelize-over-seeds scheme.

3.1. A Data-Parallel Formulation of Particle Advection

In formulating a design for particle tracing using the DPP in VTK-m, our primary task was to determine the definition for the elementary unit of work. This elementary work unit can then be mapped onto

```
INPUTS:
Seeds, vectorField, numberOfSteps, stepSize
OUTPUTS:
Advected

Advected = map<Advect>(Seeds,
                       vectorField,
                       numberOfSteps,
                       stepSize)
```

Figure 2: *Pseudo code for our implementation of particle advection using DPP. The Advect function is defined in Figure 1. It serves as the functor to the map DPP which calls it – in parallel – for all seeds. The DPP is shown in the form of primitive<functor>(arguments).*

the set of execution threads to perform the total amount of work using massive parallelism. The most natural elemental unit of work is the advection of a single particle. However there are subtleties in providing a precise definition. The traditional option, and the one we selected, is to define the unit of work as the entire advection of an individual particle. Other options include those studied in [SP16] where the unit of work is defined as a fixed number of advection steps for an individual particle, or a group of particles. These other options provide smaller granularity which provide opportunities for better load balancing of total work. This, however, comes at the cost of increased overhead for scheduling.

For our study, we decided to define the entire advection of a single particle as the elementary unit of work for the following reasons. First, this choice naturally encodes a map from an input position — the seed location — to an output position — the advected particle. Second, this map can directly be expressed by a data parallel primitive. Finally, since this is the most widely used approach, we felt it was imperative to thoroughly understand this method as it would inform directions for future improvements.

The pseudo-code in Figure 1 shows the implementation of our elementary unit of work, the `Advect(...)` function. The input to this function consists of a seed location, a vector field, an integration step size, and the maximum number of integration steps. The particle trajectory is computed using a numerical integration scheme. For this paper, we use the well-established 4^{th} order Runge-Kutta method. Advection terminates when a maximum number of steps is reached, or when the particle leaves the spatial domain of the vector field. To advect multiple seeds, the `Advect` kernel is applied to each individual seed.

The advection of each seed position, i.e. each loop iteration, is independent of all other seed locations. This leads to the following two observations: First, the computation for each seed can be performed in parallel without the need for any form of synchronization. Second, as stated above, this forms a basic unit of work that is to be executed per seed. Hence, we express the handling of a single seed by means of a functor that operates on an elementary piece of data — the seed location and wrap this functor as a VTK-m worklet. In order to keep the worklet description hardware-independent, it is important to note that all accesses to raw memory are encapsulated by so called array handles. In this way, the exact location of a data item in memory — specifically if it resides in host or device memory — is hidden from the worklet. This enables the flexible, automatic management of the actual memory by the underlying device adapter.

With the elementary operation of advection of an individual particle formulated in a generic fashion, the remaining task is to enable a concurrent execution for multiple particles. This is realized by means of a specific data parallel primitive: the `map` operation. In this specific case, we aim to *map* an input seed position to its eventual end position after advection. The map operation is one of the DPPs which is implemented in a highly optimized form by the VTK-m runtime. Hence its use — illustrated in Figure 2 — entails a platform-specific, parallel execution of the `Advect` functor on each particle on the underlying parallel hardware. In particular, if the runtime environment is an accelerator device, all data that is consumed or produced by the advection operation is automatically transferred to/from device memory. In this way, the decision of where the advection operation is executed is completely hidden from the developer of the worklet. This is the main reason why worklets can access data only via so called array handles (see above). Using the *map* DPP, we now have obtained a data parallel formulation of particle advection that resembles the basic parallelize-over-seeds principle.

Finally, we note that for the purposes of this study we are focused exclusively on the performance of particle advection techniques where only the final location of the seeds is computed. This is the exact formulation for analysis techniques such as FTLE, and a fundamental building block for other techniques which use the saved trajectories of seeds such as streamlines, pathlines, streamsurfaces, and Poincaré methods.

4. Performance Evaluation

In this section we discuss the experimental setup, including data sets, workloads, and hardware platforms, followed by the performance of our approach on each experiment.

4.1. Experimental Setup

In order to evaluate our work, we have selected a number of different experiments that cover a range of uses cases for particle advection. We use three different parameters for our study, which we vary independently. The first parameter is the vector field data. The types of flow structures present in a data set has a tremendous impact on the performance of an implementation, and so we capture various types of flows by using multiple data sets. The second parameter is the workload, which consists of the number of particles and the number of integration steps. The various types of vector field analysis techniques tend to use different classes of workloads, and so we have selected a set of workloads that capture the common use cases. The third and final parameter that we vary in our study is the execution hardware in order to capture performance on both CPU and GPU environments. We vary this third parameter at an even finer level of granularity by performing each test on several different types of CPUs and GPUs.

Data Figure 3 provides information on the three different data sets used in our study. The *Astro* data set contains the magnetic field surrounding a solar core collapse resulting from a supernova. This

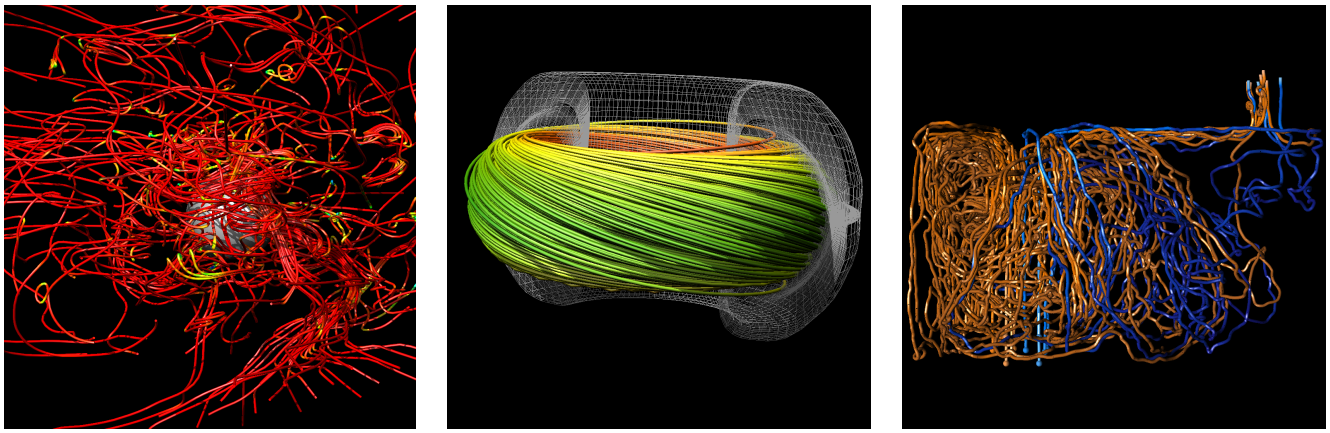


Figure 3: *Sample streamlines show typical vector field patterns in the three data sets used in this study.*

data set was generated by the GenASiS [ECBM10] code which is used to model the mechanisms operating during these solar core collapse events. The *Fusion* data set contains the magnetic field in a plasma within a fusion tokamak device. This data set was generated by the NIMROD [SGG*04] simulation code which is used to model the behavior of burning plasma. The plasma is driven in large measure by the magnetic field within the device. Finally, the *Thermal Hydraulics* data set contains the fluid flow field inside a chamber when water of different temperatures is injected through a small inlet. This data set was generated by the NEK5000 [FLPS08] code which is used for the simulation computational fluid dynamics. The vector fields for all of our representative data sets are defined on uniform grids. All three data sets feature a resolution of $512 \times 512 \times 512$ accounting for $1,536$MB per vector field. We have specifically chosen the simplest type of data representation to exclude additional performance complexities that can manifest with more complex grid types, e.g., point location in unstructured meshes.

Workloads As stated above, a workload consists of a set of particles and the number integration steps to be taken. The particles are randomly distributed throughout the spatial extents of the data set grid. We chose a set of five workloads that we felt mimicked the behavior of common uses cases, e.g., from streamlines to FTLE computations. Further, we specifically choose this set based on work by Camp et al. [CKP*13], who studied the performance of both CPU and GPU implementations, and identified workloads that were well suited to each architecture.

These five workloads are defined as follows:

- $\mathbf{W_1}$: 100 seeds integrated for 10 steps.
- $\mathbf{W_2}$: 100 seeds integrated for 2000 steps.
- $\mathbf{W_3}$: 10M seeds integrated for 10 steps.
- $\mathbf{W_4}$: 10M seeds integrated for 100 steps.
- $\mathbf{W_5}$: 10M seeds integrated for 1000 steps.

Hardware The execution environment for our study consists of the three systems deployed at the Oak Ridge Leadership Compute Facility (OLCF) (c.f. Table 1).

Table 1: *Hardware used in study.*

Machine	CPU	GPU
Rhea Partion 1	Dual Intel Xeon E5-2650 "Ivy Bridge", 2.0 GHz 16 total cores 128 GB RAM	*None*
Rhea Partition 2	Dual Intel Xeon E5-2695 v3 "Haswell", 2.3 GHz 28 total cores 1 TB RAM	2x NVIDIA K80 12 GB Memory
Titan	*Not used*	NVIDIA K20X 6 GB Memory
SummitDev	Dual IBM Power8 3.5 GHz 20 total cores 256 GB RAM	4x NVIDIA P100 16 GB Memory

- Titan is a Cray XK7, and is the current production supercomputer in use at the OLCF. It contains $18,688$ compute nodes and has a peak performance of 27 petaflops.
- Rhea is a production cluster used for analysis and visualization via pre- or post-processing. It is a 512 node commodity Linux cluster that is configured in two partitions. The first partition is targeted for processing tasks requiring larger amounts of memory and/or GPUs. The nodes in its second partition do not have GPUs.
- SummitDev is an early access 54 node system that is one generation removed from Summit, the next supercomputer that will be installed at the OLCF.

We note that both Rhea and SummitDev contain multiple GPUs on each node, however in this study we are only studying our implementation on a single GPU.

Table 2: *Timings (in seconds) for VTK-m implementations for each test in our experimental setup.*

		GPU with data transfer			GPU without data transfer			CPU		
	File	K20X	K80	P100	K20X	K80	P100	$Intel_{16}$	$Intel_{28}$	IBM $P8_{20}$
$\mathbf{W_1}$	Astro	0.627s	0.521s	0.389s	0.000s	0.011s	0.014s	0.001s	0.001s	0.001s
	Fusion	0.627s	0.521s	0.387s	0.001s	0.011s	0.015s	0.001s	0.001s	0.001s
	Thermal	0.627s	0.521s	0.392s	0.001s	0.011s	0.024s	0.001s	0.001s	0.001s
$\mathbf{W_2}$	Astro	0.648s	0.543s	0.404s	0.021s	0.033s	0.029s	0.071s	0.046s	0.053s
	Fusion	0.649s	0.543s	0.400s	0.023s	0.033s	0.028s	0.071s	0.051s	0.052s
	Thermal	0.648s	0.541s	0.395s	0.021s	0.031s	0.027s	0.074s	0.048s	0.051s
$\mathbf{W_3}$	Astro	1.511s	0.946s	0.577s	0.884s	0.436s	0.202s	3.003s	1.257s	2.327s
	Fusion	1.509s	0.961s	0.582s	0.883s	0.451s	0.210s	2.948s	1.208s	2.609s
	Thermal	1.508s	0.945s	0.583s	0.881s	0.435s	0.215s	2.801s	1.179s	2.691s
$\mathbf{W_4}$	Astro	5.193s	2.851s	1.765s	4.566s	2.341s	1.390s	28.702s	10.688s	20.708s
	Fusion	5.327s	2.795s	1.776s	4.701s	2.285s	1.404s	26.295s	10.785s	19.949s
	Thermal	5.099s	2.785s	1.777s	4.472s	2.275s	1.409s	26.641s	11.266s	19.365s
$\mathbf{W_5}$	Astro	38.660s	23.322s	13.338s	38.033s	22.812s	12.963s	256.900s	107.806s	185.852s
	Fusion	41.116s	24.450s	13.648s	40.490s	23.940s	13.276s	272.165s	107.113s	186.455s
	Thermal	39.444s	24.153s	13.626s	38.817s	23.643s	13.258s	260.740s	106.881s	193.110s

Compilers Our VTK-m code was compiled on Rhea and Titan using the 4.8.2 version of the GCC compiler, the most stable version for Titan. For SummitDev we used the closest available version, which was 4.8.5. On Rhea and Titan we used version 7.5.18 of CUDA as it is the most stable version for Titan. On SummitDev the only version of CUDA available was version 8.0.54. On all platforms, our code was compiled using full optimization flags, `-O3`

4.2. Results

In this section we present the results from our experiments. In Section 4.2.1 we present results for our VTK-m implementation across the workloads described above and discuss the performance. We also demonstrate the parallel efficiency of our implementation for CPUs. Finally, in Section 4.2.2 we compare our implementation with two hand-coded hardware specific implementations and discuss the performances.

4.2.1. VTK-m Results

The data in Table 2 contains the runtimes for our VTK-m implementation for the cross-product of the five workloads, the three data sets, and hardware types. We ran the GPU experiments under two different scenarios, which are shown in the first two sets of three columns each. For the first scenario, we assume that the vector field data resides in host memory and has to be uploaded to the device before tracing. The timings for this scenario are shown in the set of three columns of the table labeled "GPUs with data transfer".

The second scenario assumes an in situ setting: the vector field resides on the device already, either because it has been generated there or because it is uploaded once for subsequent interactive exploration. This second scenario is representative for, e.g., an in situ change of the data's representation [ACG*14] or an exploratory visualization where particles are seeded and displayed in a highly interactive fashion [BSK*07, SBK06]. The timings for this in situ scenario are shown in the set of three columns of the table labeled "GPUs without data transfer".

For workloads with few particles (e.g., $\mathbf{W_1}$ and $\mathbf{W_2}$) the overhead for data transfers to the GPU is clearly evident. As would be expected, as more work is available for the GPUs this data transfer overhead can be better amortized over the particle advection work.

Analogous timings for various CPU settings are shown in the last three columns of Table 2.

In comparing the GPU and CPU implementations, we offer the following observations. CPUs tend to perform better than GPUs for lower seed counts. We observe this behavior in workloads $\mathbf{W_1}$ and $\mathbf{W_2}$ where a small number of seeds are advected very short and very long distances respectively. In contrast, GPUs tend to perform better with higher seed counts, and more advection steps. We observe this behavior in workloads $\mathbf{W_3}$, $\mathbf{W_4}$, and $\mathbf{W_5}$. In workload $\mathbf{W_3}$, which features a large number of particles advected for a medium number of steps, we see only moderate wins for the GPU. These observations are in line with earlier studies by Camp et al. [CGC*11, CKP*13].

We note that the particle advection source code was identical for all tests; in particular, it was not hand-tuned to any hardware platform. Hence, these results provide evidence of the portable performance of a single implementation on a broad set of execution environments.

In addition, we see the expected trends in performance across hardware families. For example, the NVIDIA P100 is faster than the K80, which in turn outperforms the K20X. We see a similar trend when comparing the times for the two different generations of Intel processors on each of the Rhea partitions. We also note that the performance on the IBM Power8 CPUs that contain 20 cores each falls between the performance of the 16 and 28 core Intel processors, which aligns with expectations.

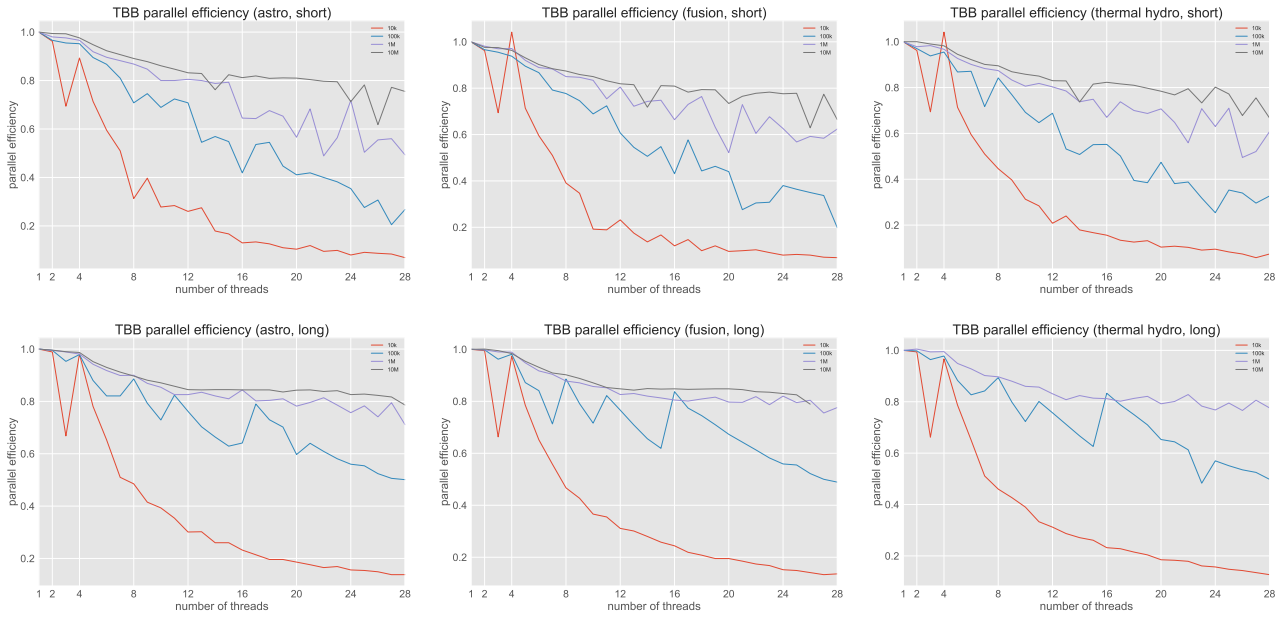


Figure 4: *Parallel efficiency for the VTK-m implementation running on the CPU of the Rhea Partition 2. The workloads shown are for* 10*k,* 100*k,* 1*M, and* 10*M seeds on all three data sets. The top row shows advections for short durations, and the bottom row shows advections for long durations. Note: Our allocation was exahusted before we could complete the data collection for the* 10*M seed case in the bottom, far right, and so these numbers are not included. Overall, we feel the scaling behavior is good, as efficiency often drops as more and more cores are used.*

Finally, we are interested in understanding the scalability of the VTK-m implementation on the CPU. Figure 4 shows the efficiency with respect to number of cores used for several workloads run on the Rhea Partition 2 CPU. For low seed counts, parallel efficiency is lacking. Quite simply, there is not enough concurrently executable work in the system to enable efficient scheduling and thus good resource utilization. However, once enough parallelism becomes available – afforded by increased number of particles – we see excellent efficiency: for short duration workloads, and populations of 1 million particles or more, parallel efficiency remains above 60%; for longer duration workloads, the efficiency is around 80%.

4.2.2. Comparisons to Other Implementations

Since our implementation is per definition agnostic of the eventual execution environment, we are particularly interested in any performance penalties due to portability. To explore these impacts, we compare our results to two different reference implementations. These comparison codes were run on the same hardware, and compiled using the same compilers as our VTK-m implementation.

First, we compare our code to hand-coded implementations for CPUs and GPUs using pthreads and CUDA, respectively. The reference code has been evaluated in [CGC*11, CKP*13].

The data in Table 3 lists the runtimes for each workload using the CUDA specific implementation, and a performance factor for the VTK-m runtimes. This factor gives the relative speed-up of our implementation over the reference implementation, i.e. factors larger than 1 indicate our implementation is faster. For many of the tests run, we observe that the VTK-m version outperforms the hand-coded implementation by factors up to $4X$.

These performance improvements are largely a function of two differences. The first difference is that VTK-m's CUDA device adapter performs all global device memory accesses through texture cache lookups. Hence, it is able to perform random accesses at a granularity of 32 bytes per load instead of 128 bytes. This gives the VTK-m implementation a significant advantage for workloads that heavily depend on highly random read operations, such as particle advection. Second, we note that the reference implementation is a more fully-featured system that can be run in a distributed memory parallel setting using several parallelization strategies (e.g., POS and POB). As such there are overheads associated with running this code on a single node using a POS approach. A combination of these factors explains the good performance of the VTK-m implementation.

We also note tests where the hand-coded implementation outperforms the VTK-m version.

First, for the Astro data set in $\mathbf{W_5}$, the hand-coded implementation performs significantly better. In the particular vector field for this data set, there are regions of the flow where particles quickly exit the grid. Rapidly terminating particles, however, induce imbalanced workloads which in turn is detrimental to overall performance. The hand-coded implementation handles these situations better than our implementation, and as a result achieves better performance for high

workloads. We see this same situation in $\mathbf{W_4}$ for the Astro data set, but since this workload consists of less work than that of $\mathbf{W_5}$ the impact of this imbalance is not nearly as dramatic.

Second, we note performance on the K80 GPU for workloads $\mathbf{W_1}$ and $\mathbf{W_2}$. For these tests the performance of the VTK-m implementation is roughly half. We suspect that the hand-coded CUDA implementation has better work management for the low seed counts of $\mathbf{W_1}$ and $\mathbf{W_2}$ on the K80 architecture, but we are at a loss to provide a specific explanation. We note that for the performance for workloads $\mathbf{W_1}, \mathbf{W_2}$ on the K80 GPU is better than the performance on the newer P100 GPU. For workload $\mathbf{W_3}$, where there are more particles, the difference in performance between the K80 and P100 is less dramatic. For workloads $\mathbf{W_4}$ and $\mathbf{W_5}$ where there is much more work, the performance maps directly onto the generation of the GPU, as expected. We note that the hand-coded GPU implementation we are using to compare was written, tuned and optimized in the time frame of the Kepler generation of GPUs. It is possible that such optimizations specific to a particular generation of hardware might not perform well on later generation hardware and need to be optimized differently.

The data in Table 4 lists the runtimes for each workload using the pthreads specific code path of the reference implementation, and a performance factor for the VTK-m runtimes. For workloads where there is less work to be done ($\mathbf{W_1}$, $\mathbf{W_2}$), the hand-coded phtreads code performs much better than the VTK-m implementation. However, it should be noted that when there are very few particles, run times are very small, and the overheads associated with each implementation tend to dominate the comparison.

In subsequent workloads, where is more work to be performed, the performance of our VTK-m implementation is comparable in many instances. We note that in general, the VTK-m implementation performs better on the Intel CPUs. This fact is not too surprising given that TBB, an Intel product, is likely optimized for Intel hardware.

One outlier in the Table 4 that is worth exploring is $\mathbf{W_5}$ for the Astro and Thermal data sets. As discussed above for the GPU implementations, the quickly exiting particles in the Astro data set leads to imbalance. We are seeing this same effect in the CPU implementation. For the Thermal data set, there is a similar issue. In the Thermal data set there are regions of the flow where particles stagnate due to zero velocities. These stagnating particles can also lead to load imbalance which in turn is detrimental to overall performance. The GPUs have enough parallelization to better amortize these stagnant particles, but in CPUs where there is less parallelization, these effects cannot be overcome. The hand-coded pthreads code handles these situations better than our VTK-m implementation, and as a result, achieves better load balancing in these situations. The issues identified in both of these data sets are planned on being addressed in the future for our VTK-m implementation.

Our final comparison is made with a fully featured production visualization and analysis software tool. The data in Table 5 contains a comparison of our VTK-m implementation to VisIt [CBB*05]. These tests were run on a CPU on the Rhea Partition 1, and compiled with the same compiler and options as our VTK-m implementation.

VisIt uses a serial execution model, and so we compare two different workloads on all three data sets using a single core execution of the VTK-m implementation. We also provide timings made with a 28 core execution for the VTK-m implementation for additional comparisons.

On the single core example, we see a clear performance increase of VTK-m that is $2-3X$ faster than the VisIt implementation across both workloads. For the 28 core example, there is not enough work in the small workload to see improvements in performance when more cores are used.

However, for the larger workload we see increased performance when using 28 cores, as is expected. We note, that VisIt is a fully featured production tool, and so there are overheads associated with the implementation in VisIt. Further, the tests run in VisIt are computing streamlines, as opposed to simply advecting seed locations. As such, there are overheads associated with the storing and managing of the particle trajectories.

4.3. Discussion

Overall, we argue that our results support our aim of creating a performance-portable formulation for particle tracing. Across all workloads, our implementation usually outperforms the reference codes. In rare cases it takes around twice the time to complete a given benchmark, with the worst case scenario ($\mathbf{W_5}$, Astro on Power8) taking approximately $3.5\times$ as long as the reference implementation. We note that for most of these cases we understand the reasons for the performance, and are planning to address these cases in the future.

Moreover, our implementation — by virtue of the underlying VTK-m runtime — shows good scaling on multi-core, shared memory machines. However, during preliminary experiments we discovered an aspect that affected both the TBB and the CUDA backends of VTK-m. Specifically, we noticed that the performance behavior of our test workloads was susceptible to changes in the underlying runtime's granularity settings: for TBB this would be the `grainsize` whereas for CUDA it would be the `blocksize` parameter for 1D scheduling. Changing both had a significant effect on performance. For the TBB device adapter, our experiments helped inform the decision in favor of a new, smaller default setting in VTK-m. In contrast, the general performance impact of the CUDA `blocksize` parameter on workloads outside particle advection is harder to assess and may require solutions like autotuning.

Beyond such technical issues, we observe that the performance of our implementation across the five chosen workloads matches the findings in published results that studied performance of hand coded CPU and GPU implementations. We therefore conclude that it is mostly bounded by the same limitations as the reference code. This gives us confidence that the portability of our VTK-m implementation is not introducing significant overhead issues. Further, we restate that our implementation does not contain the platform-specific optimizations that are typically found in hand-coded, hardware-specific implementations.

In summary, our findings suggest that our implementation shows competitive performance across a variety of hardware architectures and workloads. It therefore provides a solid, and efficient basis for more sophisticated, advection-based visualization methods.

Table 3: *Timings (in seconds) for the GPU comparison implementation along with a performance factor for the VTK-m timings (factors $>$ 1X indicate VTK-m is faster).*

		CUDA Code			VTK-m Comparison		
	File	K20X	K80	P100	K20X	K80	P100
$\mathbf{W_1}$	Astro	0.844s	0.285s	0.836s	1.35X	0.55X	2.15X
	Fusion	0.845s	0.284s	0.838s	1.35X	0.55X	2.17X
	Thermal	0.844s	0.284s	0.837s	1.35X	0.54X	2.14X
$\mathbf{W_2}$	Astro	0.869s	0.301s	0.842s	1.34X	0.55X	2.08X
	Fusion	0.874s	0.304s	0.845s	1.35X	0.56X	2.11X
	Thermal	0.871s	0.304s	0.844s	1.34X	0.56X	2.14X
$\mathbf{W_3}$	Astro	3.418s	1.959s	2.353s	2.26X	2.07X	4.08X
	Fusion	3.367s	1.824s	2.219s	2.23X	1.90X	3.81X
	Thermal	3.327s	1.856s	2.247s	2.21X	1.96X	3.85X
$\mathbf{W_4}$	Astro	6.682s	5.067s	3.564s	1.29X	1.78X	2.02X
	Fusion	8.803s	6.763s	4.420s	1.65X	2.42X	2.49X
	Thermal	8.793s	6.830s	4.500s	1.72X	2.45X	2.53X
$\mathbf{W_5}$	Astro	14.353s	12.963s	6.464s	0.37X	0.56X	0.48X
	Fusion	63.993s	54.670s	25.694s	1.56X	2.24X	1.88X
	Thermal	56.133s	49.172s	23.161s	1.42X	2.04X	1.70X

Table 4: *Timings (in seconds) for the CPU comparison implementation along with a comparison factor to the VTK-m timings (factors $>$ 1X indicate VTK-m is faster).*

		pthreads Code		VTK-m Comparison	
	File	$Intel_{28}$	IBM $P8_{20}$	$Intel_{28}$	IBM $P8_{20}$
$\mathbf{W_1}$	Astro	0.0006s	0.0002s	0.59X	0.17X
	Fusion	0.0004s	0.0001s	0.43X	0.09X
	Thermal	0.0004s	0.0001s	0.43X	0.07X
$\mathbf{W_2}$	Astro	0.001s	0.003s	0.03X	0.05X
	Fusion	0.003s	0.012s	0.05X	0.22X
	Thermal	0.001s	0.006s	0.03X	0.12X
$\mathbf{W_3}$	Astro	2.001s	2.408s	1.59X	1.03X
	Fusion	1.389s	2.137s	1.15X	0.82X
	Thermal	1.048s	1.719s	0.89X	0.64X
$\mathbf{W_4}$	Astro	11.675s	14.227s	1.09X	0.69X
	Fusion	11.247s	18.076s	1.04X	0.91X
	Thermal	8.123s	14.633s	0.72X	0.76X
$\mathbf{W_5}$	Astro	38.693s	53.015s	0.36X	0.29X
	Fusion	84.129s	156.735s	0.79X	0.84X
	Thermal	54.591s	113.881s	0.51X	0.59X

Table 5: *Timings (in seconds) for the VisIt implementations on two different workloads along with a comparison factor to the VTK-m timings (factors $>$ 1X indicate VTK-m is faster).*

			VTK-m	
	File	VisIt	1 core	28 core
100 Seeds 1000 Steps	Astro	0.0543s	2.36X	2.36X
	Fusion	0.0855s	3.56X	3.56X
	Thermal	0.0628s	2.61X	2.61X
10,000 Seeds 1000 Steps	Astro	5.5253s	2.46X	8.55X
	Fusion	7.9353s	3.22X	12.53X
	Thermal	5.9484s	2.45X	9.64X

5. Conclusion & Future Work

In this paper, we have introduced a performance-portable, general purpose formulation for one of the fundamental techniques for the analysis and visualization of vector fields: particle advection. Our implementation is based on the VTK-m framework, which has been developed to address the rapidly changing landscape of execution environments in HPC systems. The issue of portable performance across diverse architectures is of growing importance to simulation and experimental scientists across a wide set of disciplines. The growing compute and I/O imbalance in current and future HPC systems is causing a keen interest in scenarios where compute is moved to the data, as opposed to the traditional model where data are moved to the computational resources. Portability is extremely important in these use cases: a visualization code has to run with reasonable efficiency close to the data, regardless of the specific hardware that generated said data.

We have demonstrated the portable performance of our implementation on a set of typical workloads, across a representative set of vector fields, and on a diverse set of CPU and GPU hardware. We have shown that the behavior of our portable implementation agrees with previously published results on hand-coded GPU and CPU implementations. We have also compared the performance of our portable implementation to several hand-coded implementations on a variety of workloads and hardware configurations.

As stated previously, we have not performed any hand-tuning of the VTK-m backend to achieve portable performance. However, we believe that there are improvements that could be made to the VTK-m backends that would yield increased performance for particle tracing methods. We plan to explore and evaluate these aspects

and balance them against the general performance of the VTK-m backend as a whole.

Finally, there are a large number of extensions to this work, including support for streamlines where particle trajectories are stored. Because of early termination of particles, it is unknown at runtime what memory resources are required for the particle trajectories. In the future, we plan to explore methods to efficiently support storing particle trajectories for streamlines. We are also planning on support for time-varying vector fields and the analysis techniques associated with these types of vector fields. Eventually, we would like to re-evaluate performance portability for both of these cases; due to the dynamic memory allocations (streamlines) and the increased read-bandwidth requirements (pathlines) we might find different effects. In that regard, we plan to study the impact of unified memory available on new GPUs for particle tracing in general, but also to handle the storing of particle trajectories for streamlines and pathlines.

In summary, we believe that the portable performance of our implementation makes it a fruitful platform for work in particle-advection based techniques.

6. Acknowedgements

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. 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 material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Award Number 14-017566. Dr. Bernd Hentschel gratefully acknowledges the support through RWTH Aachen's Theodore von Kármán Fellowship for Outgoing Scientists. The authors would like to thank the reviewers for their careful reviews, and suggestions for this paper.

References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-Scale Data Visualization using Parallel Data Streaming. *IEEE Computer Graphics and Applications 21*, 4 (2001), 34–41. 2

[ACG*14] AGRANOVSKY A., CAMP D., GARTH C., BETHEL E. W., JOY K. I., CHILDS H.: Improved post hoc flow analysis via lagrangian representations. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization* (2014), pp. 67–75. 1, 6

[Aya15] AYACHIT U.: *The ParaView Guide – ParaView 4.3*. Kitware Inc., 2015. 2

[BH11] BELL N., HOBEROCK J.: Thrust – a productivity-oriented library for cuda. In *GPU Computing Gems – Jade Edition*. Morgan Kaufmann, 2011, pp. 359–371. 2

[Ble90] BLELLOCH G. E.: *Vector Models for Data-Parallel Computing*. MIT Press, 1990. 2

[BSK*07] BÜRGER K., SCHNEIDER J., KONDRATIEVA P., KRÜGER J., WESTERMANN R.: Interactive Visual Exploration of Unsteady 3D Flows. In *Proceedings of EG/IEEE VGTC Symposium on Visualization* (2007), pp. 251–258. 1, 3, 6

[CBB*05] CHILDS H., BRUGGER E., BONNELL K., MEREDITH J., MILLER M., WHITLOCK B., MAX N. L.: A Contract Based System for Large Data Visualizations. In *Proceedings ofIEEE Visualization* (2005), IEEE, pp. 191–198. doi:10.1109/VISUAL.2005.1532795. 2, 8

[CBP*14] CHILDS H., BIERSDORFF S., POLIAKOFF D., CAMP D., MALONY A. D.: Particle advection performance over varied architectures and workloads. In *Proceedings of the International Conference on High Performance Computing (HiPC)* (2014), pp. 1–10. 3

[CGC*11] CAMP D., GARTH C., CHILDS H., PUGMIRE D., JOY K. I.: Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Transactions on Visualization and Computer Graphics 17*, 11 (2011), 1702–1713. doi:10.1109/TVCG.2010.259. 1, 2, 6, 7

[CKP*13] CAMP D., KRISHNAN H., PUGMIRE D., GARTH C., JOHNSON I., BETHEL E. W., JOY K. I., CHILDS H.: GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization* (2013), pp. 1–8. 1, 2, 3, 5, 6, 7

[CL93] CABRAL B., LEEDOM L.: Imaging Vector Fields Using Line Integral Convolution. In *Proceedings of the ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (1993), pp. 263–270. 1

[CSH16] CHEN M., SHADDEN S. C., HART J. C.: Fast coherent particle advection through time-varying unstructured flow datasets. *IEEE Transactions on Visualization and Computer Graphics 22*, 8 (Aug. 2016), 1960–1973. doi:10.1109/TVCG.2015.2476795. 1, 3

[EBRI09] EVERTS M. H., BEKKER H., ROERDINK J. B., ISENBERG T.: Depth-dependent halos: Illustrative rendering of dense line data. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1299–1306. doi:10.1109/TVCG.2009.138. 1

[ECBM10] ENDEVE E., CARDALL C. Y., BUDIARDJA R. D., MEZZACAPPA A.: Generation of Magnetic Fields By the Stationary Accretion Shock Instability. *The Astrophysical Journal 713*, 2 (2010), 1219–1243. 5

[FBTW10] FERSTL F., BURGER K., THEISEL H., WESTERMANN R.: Interactive Separating Streak Surfaces. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1569–1577. doi:10.1109/TVCG.2010.169. 1

[FLPS08] FISCHER P., LOTTES J., POINTER D., SIEGEL A.: Petascale Algorithms for Reactor Hydrodynamics. *Journal of Physics: Conference Series 125* (2008), 1–5. 5

[GGTH07] GARTH C., GERHARDT F., TRICOCHE X., HAGEN H.: Efficient Computation and Visualization of Coherent Structures in Fluid Flow Applications. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1464–1471. doi:10.1109/TVCG.2007.70551. 1

[GKT*08] GARTH C., KRISHNAN H., TRICOCHE X., BOBACH T., JOY K. I.: Generation of Accurate Integral Surfaces in Time-Dependent Vector Fields. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1404–1411. doi:10.1109/TVCG.2008.133. 1

[Hal01] HALLER G.: Distinguished Material Surfaces and Coherent Structures in Three-Dimensional Fluid Flows. *Physica D: Nonlinear Phenomena 149*, 4 (2001), 248–277. doi:10.1016/S0167-2789(00)00199-8. 1

[HGK*15] HENTSCHEL B., GÖBBERT J. H., KLEMM M., SPRINGER P., SCHNORR A., KUHLEN T. W.: Packet-Oriented Streamline Tracing on Modern SIMD Architectures. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization* (2015), pp. 43–52. 3

[Hul92] HULTQUIST J. P.: Constructing Stream Surfaces in Steady 3D Vector Fields. In *Proceedings ofIEEE Visualization* (1992), pp. 171–178. URL: http://dl.acm.org/citation.cfm?id=949685.949718. 1

[KKKW05] KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics 11*, 6 (2005), 744–756. doi:10.1109/TVCG.2005.87. 3

[KWA*11] KENDALL W., WANG J., ALLEN M., PETERKA T., HUANG J., ERICKSON D.: Simplified parallel domain traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 10:1–10:11. URL: http://doi.acm.org/10.1145/2063384.2063397, doi:10.1145/2063384.2063397. 2

[LEG*08] LARAMEE R. S., ERLEBACHER G., GARTH C., SCHAFHITZEL T., THEISEL H., TRICOCHE X., WEINKAUF T., WEISKOPF D.: Applications of texture-based flow visualization. *Engineering Applications of Computational Fluid Mechanics 2*, 3 (2008), 264–274. 1

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P., MEREDITH J., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization* (2015), pp. 53–62. 1, 2

[LMNC15] LARSEN M., MEREDITH J., NAVRÁTIL P., CHILDS H.: Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium* (Hangzhou, China, 2015), pp. 279–286. 1, 2

[LSA12] LO L., SEWELL C., AHRENS J.: PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization* (2012). doi:10.2312/EGPGV/EGPGV12/011-020. 2

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax Toolkit: A proposed framework for data analysis and visualization at Extreme Scale. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization* (2011), pp. 97–104. doi:10.1109/LDAV.2011.6092323. 2

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: EAVL: The Extreme-scale Analysis and Visualization Library. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization* (2012). doi:10.2312/EGPGV/EGPGV12/021-030. 2

[MCHG13] MÜLLER C., CAMP D., HENTSCHEL B., GARTH C.: Distributed Parallel Particle Advection using Work Requesting. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization* (2013), pp. 1–6. 1, 3

[MLP*10] MCLOUGHLIN T., LARAMEE R. S., PEIKERT R., POST F. H., CHEN M.: Over Two Decades of Integration-Based, Geometric Flow Visualization. *Computer Graphics Forum 29*, 6 (2010), 1807–1829. 2

[MSU*16] MORELAND K., SEWELL C., USHER W., TA LO L., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., LARSEN M., CHEN C.-M., MAYNARD R., GEVECI B.: VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications 36* (2016), 48–58. 1, 2

[NLL*12] NOUANESENGSY B., LEE T.-Y., LU K., SHEN H.-W., PETERKA T.: Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), pp. 1–11. doi:10.1109/SC.2012.93. 1, 2

[NLS11] NOUANESENGSY B., LEE T.-Y., SHEN H.-W.: Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 1785–1794. doi:10.1109/TVCG.2011.219. 1, 2

[PCG*09] PUGMIRE D., CHILDS H., GARTH C., AHERN S., WEBER G.: Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis* (2009). 1, 2

[PPG12] PUGMIRE D., PETERKA T., GARTH C.: Parallel integral curves. In *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, Bethel E. W., Childs H., Hansen C. D., (Eds.). Chapman & Hall, 2012, ch. 6, pp. 91–114. 2

[SBK06] SCHIRSKI M., BISCHOF C., KUHLEN T.: Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2006* (2006), pp. 153–160. 3, 6

[SGG*04] SOVINEC C., GLASSER A., GIANAKON T., BARNES D., NEBEL R., KRUGER S., PLIMPTON S., TARDITI A., CHU M., THE NIMROD TEAM: Nonlinear Magnetohydrodynamics with High-order Finite Elements. *J. Comp. Phys. 195* (2004), 355. 5

[SKH*05] SCHIRSKI M., KUHLEN T., HOPP M., ADOMEIT P., PISCHINGER S., BISCHOF C.: Virtual Tubelets - Efficiently Visualizing Large Amounts of Particle Trajectories. *Computers & Graphics 29*, 1 (2005), 17–27. 1

[SP07] SADLO F., PEIKERT R.: Efficient Visualization of Lagrangian Coherent Structures by Filtered AMR Ridge Extraction. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1456–1463. doi:10.1109/TVCG.2007.70554. 1

[SP16] SISNEROS R., PUGMIRE D.: Tuned to Terrible: A Study of Parallel Particle Advection State of the Practice. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS)* (2016). doi:10.1109/IPDPSW.2016.173. 3, 4

[vW02] VAN WIJK J. J.: Image based flow visualization. *ACM Transactions on Graphics 21*, 3 (2002), 745–754. URL: http://doi.acm.org/10.1145/566654.566646, doi:10.1145/566654.566646. 1

[Wan10] WANG L.: On Properties of Fluid Turbulence along Streamlines. *Journal of Fluid Mechanics 648* (2010), 183–203. URL: http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=7464412, doi:doi:10.1017/S0022112009993041. 1

[WP06] WANG L., PETERS N.: The Length-Scale Distribution Function of the Distance between Extremal Points in Passive Scalar Turbulence. *Journal of Fluid Mechanics 554* (2006), 457–475. doi:10.1017/S0022112006009128. 1

[YWM07] YU H., WANG C., MA K.-L.: Parallel hierarchical visualization of large time-varying 3d vector fields. In *Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis* (2007). doi:10.1145/1362622.1362655. 2

[ZGY*17] ZHANG J., GUO H., YUAN X., HONG F., PETERKA T.: Dynamic load balancing based on constrained k-d tree decomposition for parallel particle tracing. *IEEE Transactions on Visualization and Computer Graphics* (108/2017 2017). URL: http://ieeexplore.ieee.org/document/8017633/, doi:10.1109/TVCG.2017.2744059. 3