Multi-Threaded Streamline Tracing for Data-Intensive Architectures

Ming Jiang Brian Van Essen Cyrus Harrison Maya Gokhale

Lawrence Livermore National Laboratory*

ABSTRACT

Streamline tracing is an important tool used in many scientific domains for visualizing and analyzing flow fields. In this work, we examine a shared memory multi-threaded approach to streamline tracing that targets emerging data-intensive architectures. We take an in-depth look at data management strategies for streamline tracing in terms of issues, such as memory latency, bandwidth, and capacity limitations, that are applicable to future HPC platforms. We present two data management strategies for streamline tracing and evaluate their effectiveness for data-intensive architectures with locally attached Flash. We provide a comprehensive evaluation of both strategies by examining the strong and weak scaling implications of a variety of parameters. We also characterize the relationship between I/O concurrency and I/O efficiency to guide the selection of strategy based on use case. From our experiments, we find that using kernel-managed memory-map for out-of-core streamline tracing can outperform optimized user-managed cache.

Keywords: streamline tracing, memory-map, data management, out-of-core algorithms, data-intensive computing

1 INTRODUCTION

Streamline tracing is an important tool for scientific data analysis and visualization of flow fields. It is used in many scientific domains, such as aerodynamics, fusion and hydrodynamics. Streamlines provide a very powerful way to visualize the intricate structures of a flow field. For flow analysis, they are used to identify swirling features (*i.e.* vortices) that are the dominant coherent structures in turbulent flows [13, 14]. Computing streamlines represents a significant challenge for large-scale flow fields due to the irregular, data-dependent access patterns necessary for streamline integration. The difficulty stems from the data set size, seeding density and distribution, as well as the flow field complexity [22]. In many ways, generating multiple streamlines in parallel is a data-intensive problem with a low computation to data movement ratio.

In recent years, there has been a rise in hardware technologies designed for high-performance data-intensive computing to address *Big Data* problems [12, 10]. Data-intensive applications are primarily characterized by very large working sets that exceed main memory, as well as unstructured and irregular data access patterns. They are particularly affected by memory latency, bandwidth, and capacity limitations. Rapid advancements in capacity and performance of non-volatile random access memory (NVRAM) have led to new approaches for developing data-intensive architectures, *e.g.* the Gordon [7] and Catalyst [1] systems. NVRAMs are leveraged to address the diminishing capacity of system DRAM per CPU core and to tame the increasing cost of powering main memory [29]. Since these emerging trends will influence future exascale architectures, NVRAM is expected to see wider use in memory and storage hierarchies of future HPC platforms [2]. These emerging architectures will support massive on-node I/O concurrency, but design patterns and algorithmic techniques that can extract massive I/O concurrency have not yet been established.

There are several recent studies on high-performance streamline tracing methods in both parallel distributed and out-of-core contexts (see Section 2 for more details). What is currently lacking is research on how to perform streamline tracing effectively on data-intensive architectures using node-local NVRAM. The goal of this paper is to achieve a better understanding of how to optimize data movement within streamline tracing to take advantage of data-intensive architectures. We take an in-depth look at data management strategies for streamline tracing in terms of data-intensive issues, such as memory latency, bandwidth, and capacity limitations. In general, we aim to identify and exploit high-concurrency in order to maximize bandwidth and hide latency.

In our study, we examine a shared memory multi-threaded approach to streamline tracing that targets a multi-core CPU with locally attached NVRAM. We present two data management strategies, called user-managed and kernel-managed, for out-of-core streamline tracing and evaluate their effectiveness for data-intensive architectures. Our work complements existing parallel distributed methods that use a MPI-hybrid approach [6], by focusing on increasing concurrency through multi-threading within a node. More broadly, this study examines some of the data movement issues critical to in-situ [16] and in-transit [15] techniques, wherein the data either resides or is staged onto the node-local storage.

In this paper, we systematically explore how to effectively implement user-managed and kernel-managed strategies for multi-threaded streamline tracing. We present three data-intensive optimizations for our implementations that aim to increase concurrency to maximize bandwidth and hide latency. We provide a comprehensive evaluation of both data management strategies by examining the strong and weak scaling implications of a variety of parameters including: streamline length, and seeding density and distribution. We also perform a detailed analysis of the relationship between I/O concurrency and I/O efficiency to characterize the parameter space and guide the selection of strategy based on use case. These experiments demonstrate that: 1) thread oversubscription of cores in streamline tracing is an effective technique to hide I/O latency on NVRAM, 2) in many cases, small to medium sized data transfers yield better performance than large block transfers, and 3) using kernel-managed memory-map for out-of-core streamline tracing can outperform optimized user-managed cache.

2 RELATED WORK

2.1 Streamline Tracing

Streamline tracing is a type of integration-based technique for steady (time-independent) flow fields. There has been a tremendous amount of research focused on flow visualization and analysis using integration-based techniques for both steady and unsteady flow fields. McLoughlin et al. [17] provides an overview of the state-of-the-art as well as challenges in this area. Note that there are other types of integral lines for unsteady flow fields, such as pathlines and streaklines. For this paper, we focus on exploring how to effectively implement streamline tracing for data-intensive architectures.

For large-scale data, out-of-core techniques are commonly used to achieve high I/O performance to access data stored on disk [24]. For flow visualization, Ueng et al. [26] presented an approach to

*Emails: {jiang4,vanessen1,harrison37,gokhale2}@llnl.gov

IEEE Symposium on Large Data Analysis and Visualization 2014
October 9–10, Paris, France
978-1-4799-5215-1/14/$31.00 ©2014 IEEE

compute streamlines of large unstructured grids by using an octree to partition the raw data for fast data fetching. Bruckschen et al. [3] described a technique for computing particle traces in a pre-processing stage, which are then retrieved on demand during rendering. Recently, Chen et al. [9, 8] proposed a flow-guided data layout algorithm and prefetching scheme that is more I/O efficient than the Hilbert spacing filling curve layout [23]. They developed a multi-threaded system that is similar to [25], with a single I/O thread and multiple compute threads.

For parallel distributed environments, Yu et al. [30] proposed pathlets, or short pathlines, for visualizing unsteady flow fields. Chen et al. [11] proposed to distribute data blocks by using spectral clustering to segment flow fields into blocks with minimal communication overhead. Nouanesengsy et al. [19] used a graph model and non-convex quadratic programming to estimate and balance the distribution of block workloads. Peterka et al. [21] presented a study that illustrates how the workload per process can vary due to the differences among the local flow when generating pathlines. Recently, Nouanesengsy et al. [18] proposed a pipelined model to improve I/O performance for parallel particle advection by grouping and advecting particles over multiple time intervals.

In practice, a global pre-analysis and re-partitioning of the flow field data can be computationally expensive and requires extra storage. The following parallel streamline algorithms focus on working with unmodified and pre-partitioned simulation data. Pugmire et al. [22] conducted a study using different parallelization strategies. They avoided the cost of pre-processing, and chose a combination of static decomposition and out-of-core data loading. Camp et al. [6] proposed a MPI-hybrid approach that leverages the potential for shared memory on multicore nodes.

In another study, Camp et al. [5] proposed maintaining a local cache by using an extended memory hierarchy that can improve the overall performance up to 2× over a parallel file system. We extend this idea by focusing on *how* to manage the "local cache" when using PCIe-attached NVRAMs. In particular, we explore ways to optimize the data movement for two data management strategies, and provide a comprehensive, in-depth evaluation of their performance using a variety of parameters, including data representations.

2.2 Data-Intensive Computing with NVRAM

Data-intensive node architectures with direct I/O-bus-attached NVRAM are emerging in HPC to address the demands of *big* data-intensive applications and the challenges of scaling system DRAM to match the growth in number of CPU cores [12, 28]. Current NVRAM technologies such as Flash arrays, and future technologies, such as phase-change memory (PCM) or memristors, have high bandwidth, low latency access for both sequential and random read/write operations, but require high degrees of concurrency to attain good performance.

Recent work by Van Essen et al. [27] explores the impact of node-local NVRAM on several data-intensive applications: the Livermore Metagenomics Analysis Toolkit (LMAT), and a large-scale asynchronous graph analysis library (HavoqGT). For these applications, they combine a custom memory-map runtime and node-local NVRAM to demonstrate that increasing I/O concurrency can have a dramatic improvement on the application's performance over the standard Linux memory-map runtime.

3 Data-Intensive Streamline Tracing

In this section, we take an in-depth look at the data management strategies for data-intensive streamline tracing and discuss the design choices for our evaluation.

3.1 Data Representation Schemes

As shown in Section 2.1, streamline tracing can be performed directly on the unmodified and pre-partitioned simulation data, which we refer to as native data blocks, or using a data layout optimized for spatial locality. While the optimized data layout approach often requires a computationally expensive pre-processing step, it can be significantly more efficient than native data blocks during streamline integration. Determining which approach is preferable depends on the underlying use case.

In many simulations, the regions of interest in a flow field are not known *a priori*. In these cases, *exploratory analysis* is used with the goal of constructing streamlines that characterize the overall nature of the flow and identify any regions or features that merit further analysis. For exploratory analysis, using the native data blocks is common, especially when an optimized data layout requires a pre-processing step that can take an order of magnitude more time than the actual analysis. In cases where the nature of the flow field is understood *a priori* to a certain degree, an *in-depth analysis* is used to extract information about specific flow features or regions of interest. In these cases, pre-processing the data set to optimize the data access for flow analysis can lead to overall performance gains.

3.1.1 Native Data Blocks

In this paper, we focus our evaluation on flow fields defined on structured grids with an implicit grid topology. Typically, these grids are uniformly pre-partitioned into sub-grids (data blocks) for the simulation. Rather than using the original simulation output, we use the velocity vectors that are extracted and stored in row-major order as a brick-of-values (BOV) file for each data block. During data access, a BOV file is retrieved from the node-local NVRAM if a velocity vector that resides in that data block is needed.

3.1.2 Optimized Data Layout

Although there are several optimized data layouts for streamline tracing, we chose the Hilbert space filling curve [23] for the following reasons. First, the Hilbert curve is a popular technique that has been used in numerous applications in large-scale visualization and analysis. Second, compared to data-aware layouts, such as the flow-guided layout [9, 8], there are fewer parameters that a user would need to tune in order to use the Hilbert curve effectively. Third, it exhibits optimal geometric locality properties for efficient data traversal [23]. Fourth, its implementation for structured grids is relatively straightforward, and its index computation has a negligible impact on the overall performance for data-intensive applications.

3.2 Data Management Strategies

Effectively managing the data is a critical aspect of optimizing data movement within streamline tracing for data-intensive architectures. Our goal is to identify and exploit high I/O concurrency in order to maximize bandwidth and hide latency. We present two data management strategies for multi-threaded streamline tracing: user-managed and kernel-managed. In order to capture the data movement of existing streamline tracing methods, we provide concrete implementations for both strategies using the two data representation schemes described previously: native data blocks and optimized data layout. Our four implementations are thus referred to as: native-user, hilbert-user, native-`mmap` and hilbert-`mmap`.

Figure 1 illustrates the data layout schemes in file and the corresponding data access patterns in memory for all four implementations. To isolate the characteristics of NVRAM for our evaluation, we assume all data of interest have been constructed and transferred onto the node-local storage before streamline integration. Details on the construction and transfer times are provided in Section 5.

The key differences between the two data management strategies are: 1) how data movement (I/O) is handled between NVRAM and memory, and 2) who (user or kernel) manages cache occupancy and capacity once data is moved from NVRAM into memory. Figure 2 illustrates three different types of I/O available on the Linux OS. Managing cache occupancy requires determining if the desired data

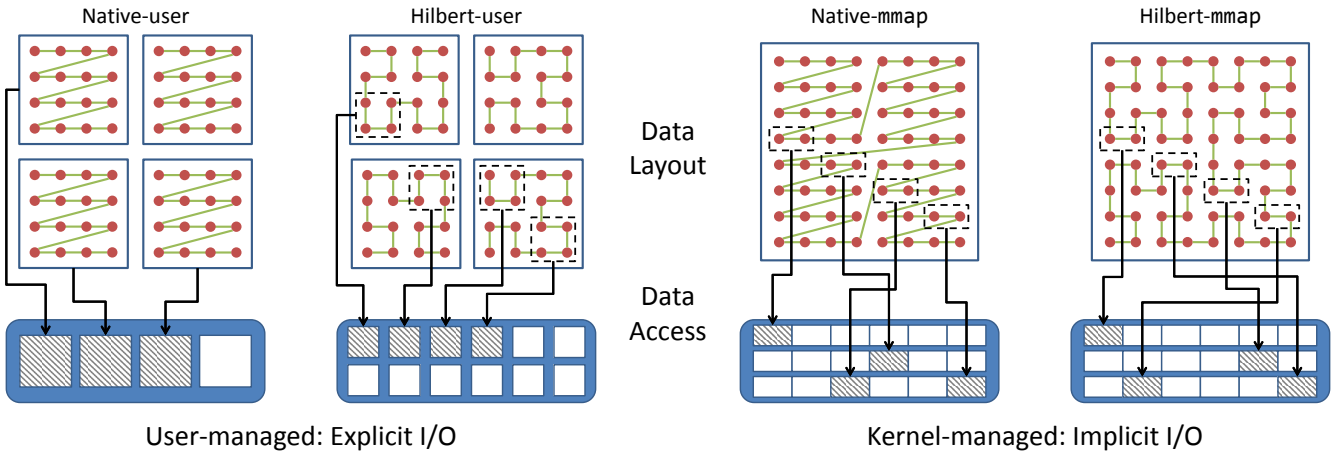


Figure 1: Illustrates the data layout scheme in file and the corresponding data access pattern in memory for each management implementation.

is in the cache or has to be fetched from storage. Managing cache capacity requires determining what data to evict from the cache. Additionally, these actions have to be performed safely in a shared memory multi-threaded environment.

3.2.1 User-Managed Strategy

The user-managed strategy relies on explicit I/O with a user space cache to buffer portions of the data set as they are accessed during streamline integration. It is utilized in existing streamline tracing methods and is typically implemented as a "least recently used (LRU) cache" [22, 5, 6, 9]. For this strategy, a read request can use either standard I/O, which reads the data into the system's page cache and then copies it into the user cache, or direct I/O, which is not buffered by the OS. Our user-managed cache implementation uses a LRU eviction policy and locks to provide safe access and management of the cache.

For our *native-user* implementation, data access is performed using the native data blocks. No extra processing is required to handle these data blocks. For our *hilbert-user* implementation, once the original data set has been serialized into the Hilbert curve, the serialized data can be divided into various chunk sizes and grouped into multiple files. Data access requires a Hilbert index conversion and is performed at the chunk level, which can be much smaller than the native data blocks. We provide a detailed analysis of the implication of using different chunk sizes in terms of I/O concurrency and I/O efficiency in Section 6.2.

3.2.2 Kernel-Managed Strategy

The kernel-managed strategy relies on implicit I/O through the OS virtual file system buffer cache algorithms that can be interfaced using the memory-map (`mmap`) functionality. When using `mmap` I/O, the application accesses the file data as if it were in memory through load and store memory requests. All data movement into the system's page cache is handled implicitly by the OS and all data is moved in page-sized blocks (*e.g.* typically 4KiB). Using the kernel-managed `mmap` runtime relieves the user from most aspects of data management. Data within the page cache is evicted using a least recently faulted (LRF) policy that approximates LRU, and occupancy is determined through a combination of dedicated hardware and low-level system software.

Using `mmap` I/O for implementing out-of-core algorithms has recently become a viable option for data-intensive architectures with node-local NVRAM [20, 27]. The benefit of the `mmap` interface is that access to the `mmap`-ed region in the application's address space will transparently check if the page is in memory and if not, it will trigger the necessary file I/O for the data. Another benefit is that a page is shared by all processes that access data on that page.

For our *native-`mmap`* implementation, we considered constructing the data into a single file of either full row-major order or block row-major order. Although the full row-major order requires no additional index conversion during data access, our tests indicated that it performs worse during data construction and access due to poor spatial locality. In our evaluation, we use the block row-major order as illustrated in Figure 1. For our *hilbert-`mmap`* implementation, we used a single file for the serialized data in Hilbert layout.

4 Evaluation Methodology

In this section, we present our methodology for evaluating the performance of the data management strategies for shared memory multi-threaded streamline tracing. Our evaluation includes: data intensive optimizations, I/O concurrency and I/O efficiency trade-off, and strong and weak scaling implications.

4.1 Data-Intensive Optimizations

The first optimization deals with how data movement (I/O) is handled for the user-managed strategy. As presented in Section 3.2.1, a read request for this strategy can use either standard I/O (buffered) or direct I/O (unbuffered). For existing streamline tracing methods using this strategy, the issue of *standard I/O vs. direct I/O* has been mostly ignored. Standard I/O allows for unaligned data accesses, but requires a second copy from the page cache into the application's memory buffer and doubles the amount of memory necessary. Direct I/O is much more efficient, but has the requirement that I/O accesses have to be aligned to file blocks. Detailed analysis on this optimization is shown in Section 6.1.1.

The second optimization deals with *minimizing lock contention* on the user-managed cache to increase I/O concurrency. Existing out-of-core streamline tracing methods [9, 8] use an approach (single I/O thread and multiple compute-threads) that was designed for the traditional "spinning disk". Modern NVRAM has a much higher bandwidth that cannot be saturated with a single I/O thread. With multiple I/O threads, lock contention on the cache can degrade performance. In Section 6.1.2, we explore the benefits of splitting the cache into independent banks to reduce lock contention, at the cost of increasing cache fragmentation.

The third optimization deals with *managing page cache capacity* for the kernel-managed strategy. One of the challenges for managing an out-of-core `mmap` implementation is that the Linux runtime prefers not to evict `mmap`-ed data, which as observed in [28] can degrade performance. We take advantage of the fact that for streamline tracing the data is read-only, thus we can use the Linux `madvise(-,-,MADV_DONTNEED)` system call to indicate that any of the data can be evicted. The performance impact of this optimization is shown in Section 6.1.3.

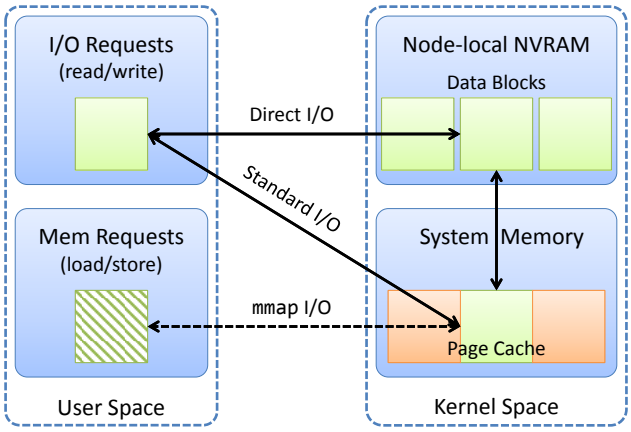


Figure 2: Linux OS I/O: standard, direct and `mmap`. Solid arrow indicates data copying and dashed arrow indicates data mapping.

4.2 I/O Concurrency and I/O Efficiency

Optimizing data movement for NVRAM requires a balance between available I/O concurrency and I/O efficiency. Traditionally, large I/O transfers have been more efficient because they can maximize the utilization of the I/O bus bandwidth with relatively low concurrency. However, NVRAM technologies support much higher levels of I/O concurrency and thus can saturate the I/O bus bandwidth with many, smaller I/O transfers.

For the data management strategies, there are a variety of transfer sizes used during data access. Whereas the transfer size for kernel-managed strategy is at the page size (*e.g.* typically 4KiB), the transfer size for user-managed strategy varies from the native block size to Hilbert chunk size, both of which can be orders of magnitude larger than a page. Hence, one way to determine which strategy is effective for a given use case is by characterizing the relationship between I/O concurrency and I/O efficiency in terms of transfer size. To accomplish this, we designed an experiment using the hilbert-user implementation, which allows us to vary the chunk size at runtime without the need to re-construct the data layout. Results from this experiment are provided in Section 6.2.

4.3 Scaling: Strong and Weak

We compare the performance of the data management strategies using strong and weak scaling experiments by varying the following parameters: streamline length (maximum number of integration steps), and seeding density and distribution. We use both uniform seeding (exploratory analysis) and feature-based seeding (in-depth analysis) at various levels of density. For strong scaling, we use a fixed streamline length at 8192 and double the number of threads from 2 to 64. For weak scaling, existing work (see Section 2) vary the streamline length, seeding density or both. From a problem-size perspective, doubling the streamline length is equivalent to doubling the seeding density. For our experiments, we decided to double the streamline length along with the number of threads.

For uniform seeding, we uniformly distribute the seed points throughout the data set, starting with one seed point per native data block. We increase the seeding density by doubling in each dimension. For feature-based seeding, we utilize high vorticity magnitude ($|\omega|$) as an indicator for the presence of vortical flows, starting with an initial seed set at high threshold. We increase the seeding density by using lower threshold ranges.

5 Experimental Setup

5.1 Implementation Details

Our streamline tracing code is implemented using the fourth-order Runge-Kutta numerical integration scheme with a constant step size. Other commonly used integration schemes can be utilized in our evaluation just as well. For the user-managed strategy, we implemented a pre-allocation scheme for the LRU cache rather than a typical lazy allocation scheme. Although lazy allocation seems reasonable for standard I/O, it can lead to excessive memory fragmentation for direct I/O due to page-alignment. Effectively, "pointers" to page-aligned buffers would each take up an entire page. For small data blocks, an extra page of overhead per block can noticeably reduce the overall capacity of the cache. For the kernel-managed strategy, the implementation was much simpler. Its mapping step is implemented using the `mmap` system call, which requires no additional index conversion during data access. For multi-threading through OpenMP, user-managed requires extra logic to ensure thread-safety, whereas kernel-managed does not.

5.2 Data-Intensive HPC System

We used the LLNL Hyperion Data-Intensive Testbed (Hyperion-DIT) to conduct our performance evaluation. Each node in the Hyperion-DIT system is a dual socket 8-core 2.67GHz Intel Xeon E5640 CPU with 24GiB of DRAM, and 4× 160GiB Fusion-io Drive Duo PCIe 1.1 x4 Flash card. Each Fusion-io Duo Flash card presents as 2 independent devices, so all 8 devices are in a RAID-0 configuration that has 640GiB of capacity with read-ahead set to 8KiB. For our experiments, we used RHEL kernel version 2.6.32 on an XFS file system. As typical of HPC systems, the Linux swap subsystem (swapping of memory to disk) has been disabled.

Table 1 provides the construction timing in seconds on the NVRAM (Fusion-io). For comparison, we also provide timings for the local solid-state drive (SSD) and the global parallel file system (Lustre). For each implementation, construction starts with native data blocks on Lustre and ends with the data layout on Fusion-io. For native-user, the timing simply refers to the copy time from Lustre onto Fusion-io. For native-`mmap`, construction can be accomplished by concatenating the native data blocks into a single file. For hilbert-user, we decided to use a single file to avoid unnecessary file I/O with multiple files. Thus its construction timing is the same as hilbert-`mmap`.

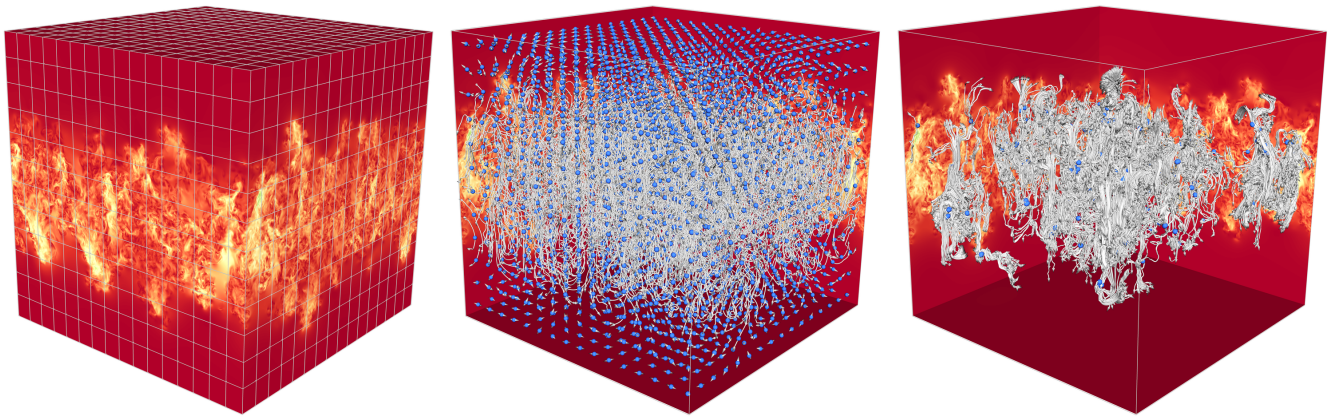
Storage	*Nat-user*	*Nat-mmap*	*Hilbert*
Fusion-io	1035	976	4559
SSD	7355	7213	7844
Lustre	1876	1834	12915

Table 1: Construction time for each implementation in seconds.

5.3 Rayleigh-Taylor Simulation

For our evaluation, we use data from a 3072^3 Rayleigh-Taylor (RT) instability simulation from LLNL [4]. As this simulation evolves, two fluids mix creating a turbulent mixing layer that yields a complex flow field. We selected a late-time snapshot from this simulation that contains a flow field that varies from steady unmixed fluid, to fast, turbulent flow in the mixing layer. Having this variation in flow speed is what makes this data ideal for conducting our experiments, because it can produce different data access patterns that are representative of a wide range of flow fields.

The grid was uniformly pre-partitioned into 16×16×12 data blocks, each of dimensions 192×192×256. The BOV file for each native data block is of size 108MiB. For the Hilbert layout construction, we decided to decompose the original 3072^3 data set into 3×3×3 partitions of size 1024^3. Within each partition, a Hilbert layout is constructed. Figure 3 shows the velocity magnitude of the data as well as streamlines traced from uniform and feature-based seedings. The uniform seeding sets span the entire grid with sizes: s1) 3072, s2) 8∗3072 and s3) 64∗3072. The feature-based seeding sets, on the other hand, reside only in the turbulent mixing layer with sizes: s1) 8509, s2) 32227 and s3) 163523, with threshold ranges: $|\omega| \geq 10$, $10 > |\omega| \geq 8.5$, and $8.5 > |\omega| \geq 7$, respectively.



(a) Velocity magnitude (b) Uniformly seeded streamlines (c) Feature-based seeded streamlines

Figure 3: Visualization of the Rayleigh-Taylor simulation using a colormap and traced streamlines from uniform and feature-based seedings.

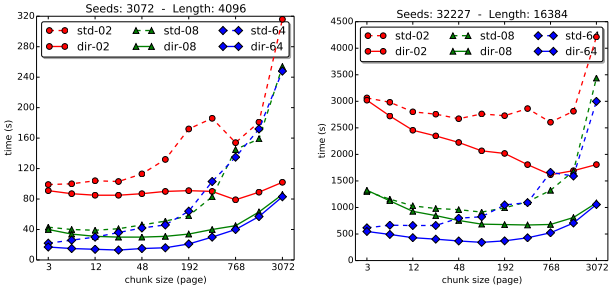
6 Results

In this section, we present our experimental results designed to evaluate the effectiveness of the data management strategies for data-intensive streamline tracing. Details on our evaluation methodology are presented in Section 4. Even though we performed a full parameter sweep for all experiments, we only present representative results for each experiment due to space considerations.

6.1 Data-Intensive Optimizations

6.1.1 Standard I/O vs. Direct I/O

As described in Section 4.1, selecting standard I/O vs. direct I/O only applies to the user-managed strategy. We are interested in understanding which I/O method is effective for streamline tracing at various data chunk sizes and levels of concurrency. For this experiment, we developed test cases using uniform (s1: 3072) and feature-based (s2: 32227) seedings with two streamline lengths: 4096 and 16384. We used hilbert-user, which allowed us to adjust the chunk size for each run from 3 to 3072 pages, and selected three levels of concurrency: 2, 8 and 64 threads.



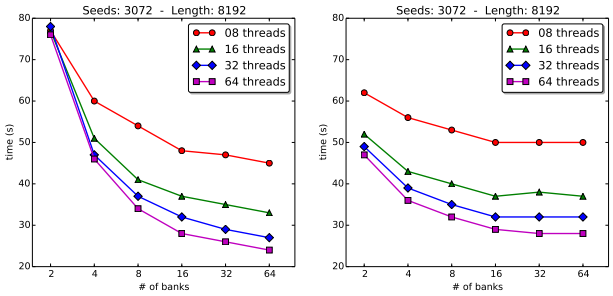
(a) Uniform-s1 – length 4096 (b) Feature-s2 – length 16384

Figure 4: Comparing standard I/O vs. direct I/O in hilbert-user using various chunk sizes with uniform and feature-based seedings.

Figure 4 shows the results for standard I/O (dashed lines) and direct I/O (solid lines). The x-axis depicts the chunk size in number of pages, and the y-axis is time to completion. *For both test cases, direct I/O performs better than standard I/O across all chunk sizes and levels of concurrency, with speedups of up to* 3.1× *and* 2.98×, *respectively.* The benefit is non-linear across chunk sizes, and the best performing chunk size is different for each test case (see Section 6.2 for more details). We note that standard I/O performs especially poorly at larger chunk sizes, due to memory pressure since a larger amount of data is duplicated in the system and user buffers than with smaller chunks. Hence, direct I/O also benefits native-user with 108MiB blocks as well.

6.1.2 Minimizing Lock Contention

Banking can be an effective technique to mitigate the cost of cache contention at high levels of concurrency for the user-managed strategy. To better understand under which conditions is banking effective for streamline tracing, we designed an experiment using two data chunk sizes (12 and 96 pages) for hilbert-user using uniform (s1: 3072) seeding at length 8192.



(a) Chunk size: 12 pages (b) Chunk size: 96 pages

Figure 5: Impact of banking hilbert-user at two chunk sizes on overall timing using uniform seeding across a range of bank sizes.

Figure 5 shows the results from four levels of concurrency: 8, 16, 32 and 64 threads. The x-axis depicts the number of banks, and the y-axis is time to completion. *For both chunk sizes, banking improves the performance across all four levels of concurrency.* However, we note that banking shows a higher improvement for the smaller chunk size than the larger one, with speedups of up to 3.17× and 1.68×, respectively. The reason for this is because for smaller chunk sizes, there is more cache activity as concurrency increases and thus more lock contention that banking can mitigate.

#Banks	*1*	*2*	*4*	*8*	*16*
Timing	408	409	412	419	438
Hits	3914961	3914903	3914762	3914339	3913244
Misses	23407	23465	23606	24029	25124

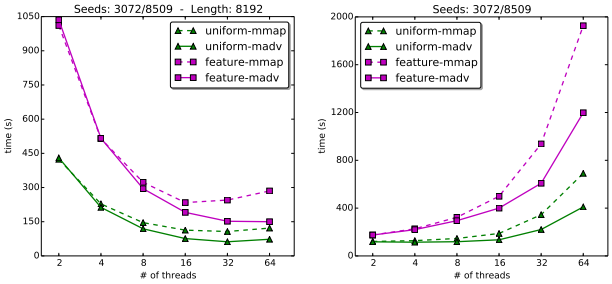
Table 2: Impact of banking native-user at 16 threads on overall timing (seconds) as well as cache hits and misses.

Note that as the number of banks increases, the improvement tapers off, and as the chunk size increases, the tapering occurs sooner at smaller bank sizes (see Figure 5b). In fact, at very large chunk sizes, banking provides no improvement. To illustrate this, we ran an experiment using native-user with 108MiB blocks. Table 2 shows the results using 16 threads and the number of banks ranging from 1 to 16. Clearly for blocks this large, the overhead of lock

contention is minimal with respect to the cost of data movement, as there is no benefit to adding banks. Furthermore, the impact of fragmenting the cache into disjoint banks reduces its effective capacity and leads to an overall slowdown, as evidenced by the cache hits and misses shown in Table 2.

6.1.3 Managing Page Cache Capacity

One challenge with using `mmap` I/O is that there is no easy way to explicitly identify which data to remove when the page cache is full. Periodically applying `madvise` with the `MADV_DONTNEED` option can reduce the performance penalty from evicting `mmap` pages. To better understand how this option can affect the performance of streamline tracing using native-`mmap` and hilbert-`mmap`, we designed an experiment using both uniform (s1: 3072) and feature-based (s1: 8509) seedings at length 8192. We also ran experiments to determine the ideal frequency to apply this option. The results showed that from 2 to 32 seconds interval, there was no noticeable change in performance. For consistency, we decided to use a 4-second interval for the following experiment.



(a) Native-`mmap` strong (b) Native-`mmap` weak

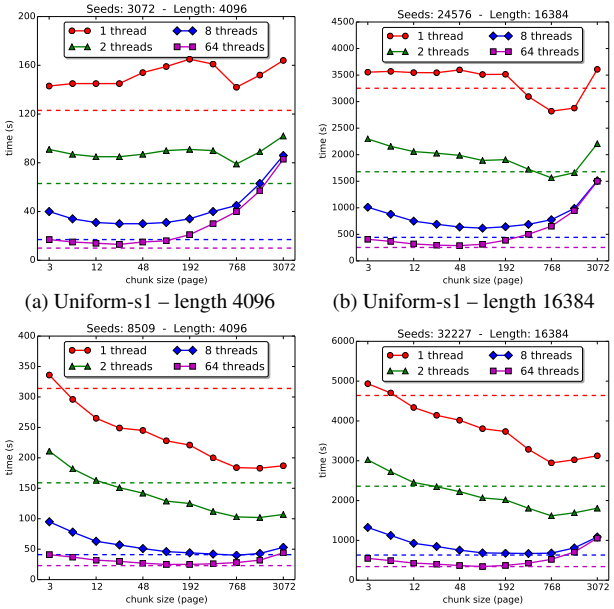
Figure 6: Comparing default `mmap` with `mmap+madvise` for managing page cache capacity using strong and weak scaling tests.

Figure 6 shows the native-`mmap` results from strong and weak scaling tests for default `mmap` (dashed lines) and `mmap+madvise` (solid lines). The x-axis depicts number of threads, and the y-axis is time to completion. *Overall, applying the `madvise` option improved the performance with speedups of up to* 1.90× *for native-`mmap` and* 1.52× *for hilbert-`mmap`.* We note that the overall performance improvement is higher for native-`mmap` than hilbert-`mmap`, because it has poorer spatial locality. Therefore, native-`mmap` has to move more data and evict more pages, which puts more pressure on the page cache eviction handler, thus giving `madvise` a greater opportunity for improvement.

6.2 I/O Concurrency and I/O Efficiency

As presented in Section 4.2, optimizing data movement requires a balance between I/O concurrency and I/O efficiency in terms of transfer size. Understanding this relationship can help us determine when a user-managed strategy is more advantageous than a kernel-managed strategy for streamline tracing. To help answer this question, we designed an experiment exploring a variety of conditions for uniform (s1: 3072) and feature-based (s1: 8509) seedings, using two streamline lengths: 4096 and 16384. In order to vary the transfer size, we used hilbert-user to adjust the data chunk size, which is its transfer size, for each run from 3 to 3072 pages.

Figure 7 shows the results from four levels of concurrency: 1, 2, 8 and 64 threads for uniform (top row) and feature-based (bottom row) seedings. The x-axis depicts the chunk size in number of pages, and the y-axis is time to completion. From these results, we observe the following. First, at lower concurrency, larger transfer sizes perform better, especially for feature-based seeding (see Figure 7c and 7d) due to spatial coherence of the resulting streamlines. Second, as concurrency increases, smaller transfer sizes perform better and show better improvement from having concurrency, with



(a) Uniform-s1 – length 4096 (b) Uniform-s1 – length 16384

(c) Feature-s1 – length 4096 (d) Feature-s1 – length 16384

Figure 7: Relationship between I/O concurrency and I/O efficiency using various chunk sizes with uniform and feature-based seedings.

speedups of up to 2.48× from 8 to 64 threads. *Finally, the optimal transfer size at higher concurrency is towards the middle, due to the tradeoff between higher transfer efficiency and data overfetch.*

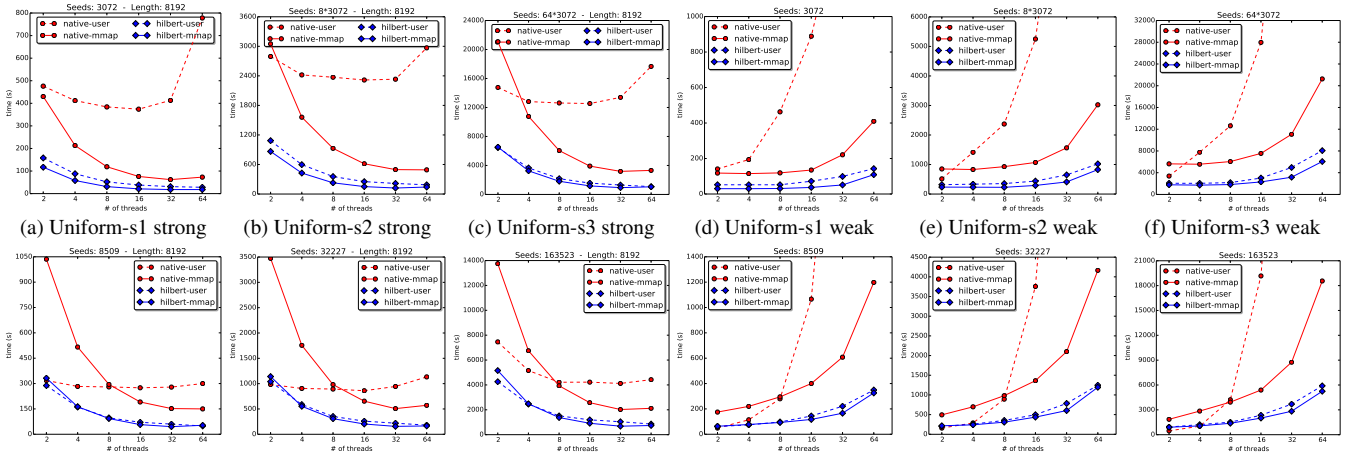
Note that where this "sweet spot" in the tradeoff occurs also depends on both the simulation data as well as the system configuration. Compounding this problem is that as I/O concurrency increases, the sweet spot for transfer size also shifts. For example, in Figure 7d, the sweet spot shifts from 784 pages at 1 thread down to 96 pages at 64 threads. This shifting is another reason why optimizing user-managed cache for streamline tracing is challenging.

Figure 7 also plots the performance of hilbert-`mmap` as a horizontal dashed line in the color corresponding to the concurrency level. The purpose is to show how `mmap`'s performance at a fixed transfer size of 1 page compares to the user-managed cache at different transfer (chunk) sizes. Our results show that at lower concurrency, hilbert-user typically performs better, especially at larger transfer sizes for feature-based seeding. At higher concurrency, however, hilbert-`mmap` is consistently as good as the best hilbert-user results, because it can manage smaller transfer sizes (1 page) more efficiently through hardware support for paging.

6.3 Scaling: Strong and Weak

We examine the strong and weak scaling implications of a variety of parameters including: streamline length, and seeding density and distribution. We compare optimized versions of user-managed and kernel-managed strategies, by leveraging results from Section 6.1, to ensure a fair comparison. For native-user, we used direct I/O with a buffer size of 160 data blocks since that gave the best overall performance for all seeding densities. For hilbert-user, we also used direct I/O with chunk size of 96 pages with bank size corresponding to the number of threads since that gave the best overall performance. For native-`mmap` and hilbert-`mmap`, we used `madvise` with the `MADV_DONTNEED` option at 4-second intervals.

Figure 8 shows the results for strong and weak scaling for uniform (top row) and feature-based (bottom row) seedings at three levels of density. The dashed lines show the user-managed results, and the solid lines show the kernel-managed results. The x-axis depicts number of threads, and the y-axis is time to completion. In order to generate legible graphs, we excluded the last two data points,



(a) Uniform-s1 strong (b) Uniform-s2 strong (c) Uniform-s3 strong (d) Uniform-s1 weak (e) Uniform-s2 weak (f) Uniform-s3 weak

(g) Feature-s1 strong (h) Feature-s2 strong (i) Feature-s3 strong (j) Feature-s1 weak (k) Feature-s2 weak (l) Feature-s3 weak

Figure 8: Strong and weak scaling tests for optimized user-managed and kernel-managed strategies using uniform and feature-based seedings.

at 32 and 64 threads, for native-user in all weak scaling cases. The runtimes for these excluded points are up to two orders of magnitude higher than the plotted points.

Overall, these results indicate that kernel-managed outperforms user-managed in both strong and weak scaling tests for both native and Hilbert representations. For native representation (red lines) under uniform seeding, native-user performs better than native-`mmap` at 2 threads, but the performance drastically degrades with increasing concurrency. Under feature-based seeding, the crossover point for native-user and native-`mmap` shifts from 2 threads to 8 threads, but overall, native-`mmap` still achieves the best performance. For Hilbert representation (blue lines), the performance difference between the highly optimized hilbert-user and hilbert-`mmap` is much smaller. At lower concurrency hilbert-user performs better, but at higher concurrency hilbert-`mmap` performs better.

For strong scaling, native-`mmap` achieves speedups of up to 4.70× over native-user, and hilbert-`mmap` of up to 1.76× over hilbert-user. For weak scaling, native-`mmap` achieves speedups of up to 7.99× over native-user, and hilbert-`mmap` of up to 1.58× over hilbert-user. Note that increasing the seeding density for both uniform and feature-based seedings does not change the overall scaling behavior, which indicates that for capturing poor weak scaling behavior, increasing streamline length is more effective.

7 Discussion

In this work, we took an in-depth look at the data management strategies for streamline tracing in terms of issues, such as memory latency, bandwidth, and capacity limitations, that are applicable to future HPC platforms. We focused on evaluating the effectiveness of user-managed and kernel-managed strategies for multi-threaded streamline tracing for data-intensive architectures. Through a series of carefully designed experiments and scaling tests, we determined under which conditions each approach (native-user, native-`mmap`, hilbert-user, and hilbert-`mmap`) performs well and explored some of the reasons behind their performance.

Native-user generally scaled the worst in both strong and weak scaling tests. In particular, increasing the streamline length is worse for native-user because computing longer streamlines requires reading more data blocks into memory without much opportunity for spatial reuse. Our results show that at higher concurrency levels smaller blocks are preferable to larger blocks. However, for shorter streamlines, which are used in exploratory analysis, native data blocks at lower concurrency sometimes performed the best.

Native-`mmap` can be a viable alternative to native-user, especially considering the minimal construction cost, in terms of time and effort, for the block row-major order. Although it does not perform as well at lower concurrency, its overall performance at higher concurrency is drastically better than native-user.

Hilbert-user can be optimized in several ways, including direct I/O and banking. Unfortunately, selecting the optimal set of parameters in conjunction with the chunk size can be a challenging task. There is no one set of parameters that yields the optimal performance under the tested conditions. For example, the ideal chunk size shifts from larger to smaller with increasing concurrency.

Hilbert-`mmap` achieved the best overall performance amongst the four approaches. In contrast to hilbert-user, it does not require tuning a complex set of parameters – applying `madvise` with the `MADV_DONTNEED` option periodically is straightforward. The advantage of using `mmap` to access data in a Hilbert layout is that it can handle smaller transfer sizes (1 page) much more efficiently through hardware paging support than hilbert-user.

In general, a user-managed strategy requires the user to actually implement additional, and often nontrivial, cache management code that must perform well and ensure thread safety. If equivalent (or better) performance can be achieved with a kernel-managed strategy, then the user can focus their time and energy on other tasks.

One shortcoming of using the kernel-managed `mmap` is the need to combine all the data blocks into a single file. Although we have demonstrated that this step can be performed efficiently, it still serves as a barrier to `mmap`'s adoption. When a file is too big to fit onto node-local NVRAM, it is conceivable to map in the file from a parallel file system. However, this approach may challenge the performance of kernel page caching. Another shortcoming is that the transfer size for the default `mmap` cannot be readily adjusted to support larger sizes, which our experiments have shown to be beneficial for hilbert-user. Additionally, the kernel-managed `mmap` does not perform as well for traditional "spinning disk" technologies that cannot support high bandwidth with low latency random access.

At a high level, our experiments demonstrate the following. First, contrary to common belief, thread oversubscription of scientific data analysis tasks, such as streamline tracing, is an effective technique to hide I/O latency on NVRAM. Second, in many cases, small to medium transfer sizes yield better performance than large block transfers. Third, using kernel-managed `mmap` for out-of-core computation, when managed properly, can outperform optimized user-managed cache. This study quantitatively overturns the well accepted notion that it is necessary to devote considerable effort to tune a user-managed cache to attain good performance.

8 Conclusion

In this work, we examined a shared memory multi-threaded approach to streamline tracing that targets data-intensive architec-

tures with node-local NVRAM. We presented two data management strategies for streamline tracing: user-managed and kernel-managed, along with data-intensive optimizations that achieved speedups of up to 3.17×. We also took an in-depth look at the relationship between I/O concurrency and I/O efficiency to better understand the key issues for optimizing data movement for NVRAM. Finally, we provided a comprehensive evaluation of both strategies by examining the strong and weak scaling implications of a variety of parameters. From our experiments, we find that using kernel-managed `mmap` for out-of-core streamline tracing can outperform optimized user-managed cache. Our scaling results for each strategy, along with the construction times, can be used together as a guide for users to select the optimal strategy based on their use case. More broadly, this study addressed some of the data movement issues critical to in-situ and in-transit techniques for visualization and analysis at extreme-scale.

For future work, we plan to apply our evaluation methodology to other types of integration-based techniques, such as pathline tracing for unsteady flow fields. While our out-of-core multi-threaded approach is able to scale to very large data sets, it is also possible to compose this approach with existing parallel distributed methods. We plan to examine such compositions through a MPI+OpenMP implementation for data-intensive architectures. As noted in Section 7, the ability to adjust the transfer size at runtime for kernel-managed `mmap` is currently inaccessible for applications. Current research on a data-intensive `mmap` (DI-MMAP) runtime [27] is exploring how to optimize for out-of-core, data-intensive computing by providing such capabilities. Using DI-MMAP with streamline (and pathline) tracing is the subject of future work.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LLNL) under Contract DE-AC52-07NA27344 (LLNL-CONF-645076). Funding partially provided by LDRD 13-ERD-025. We want to thank Bill Cabot, Andy Cook and Paul Miller at LLNL for access to the Rayleigh-Taylor simulation data set.

References

[1] Lawrence Livermore, Intel, Cray Produce Big Data Machine to Serve as Catalyst for Next-Generation HPC Clusters. https://www.llnl.gov/news/newsreleases/2013/Nov/NR-13-11-01.html, November 2013.

[2] Synergistic Challenges in Data-Intensive Science and Exascale Computing, DOE ASCAC Report. http://science.energy.gov/~/media/ascr/ascac/pdf/reports/2013/ASCAC_Data_Intensive_Computing_report_final.pdf, 2013.

[3] R. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-time out-of-core visualization of particle traces. In *IEEE symposium on parallel and large-data visualization and graphics*, pages 45–50, 2001.

[4] W. H. Cabot and A. W. Cook. Reynolds number effects on rayleigh–taylor instability with possible implications for type ia supernovae. *Nature Physics*, 2(8), 2006.

[5] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. Joy. Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *IEEE Symposium on Large Data Analysis and Visualization*, pages 57–64, 2011.

[6] 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, 2011.

[7] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 217–228, 2009.

[8] C.-M. Chen, B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Flow-guided file layout for out-of-core pathline computation. In *IEEE Symposium on Large Data Analysis and Visualization*, 2012.

[9] C.-M. Chen, L. Xu, T. Lee, H. Shen, L. Xu, T.-Y. Lee, and H.-W. Shen. A flow-guided file layout for out-of-core streamline computation. In *IEEE Pacific Visualization Symposium*, pages 145–152, 2012.

[10] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, (0), 2014.

[11] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *IEEE Pacific Visualization Symposium*, pages 87–94, 2008.

[12] M. Gokhale, J. Cohen, A. Yoo, W. Miller, A. Jacob, C. Ulmer, and R. Pearce. Hardware technologies for high-performance data-intensive computing. *Computer*, 41(4):60–68, April 2008.

[13] M. Jiang, R. Machiraju, and D. S. Thompson. Detection and Visualization of Vortices. In *Visualization Handbook*, pages 287–301. Academic Press, 2004.

[14] W. Kendall, J. Huang, and T. Peterka. Geometric quantification of features in large flow fields. *IEEE Computer Graphics and Applications*, 32(4):46–54, 2012.

[15] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *IEEE International Symposium on Parallel Distributed Processing*, 2009.

[16] K.-L. Ma. In-situ visualization at extreme scale: challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6), 2009.

[17] Y. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum*, 29, 2010.

[18] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel particle advection and ftle computation for time-varying flow fields. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 61:1–61:11, 2012.

[19] 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, 2011.

[20] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[21] 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 *IEEE International Parallel & Distributed Processing Symposium*, pages 580–591, 2011.

[22] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *International Conference on High Performance Computing Networking, Storage and Analysis*, pages 16:1–16:12, 2009.

[23] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1942.

[24] C. T. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-Core Algorithms for Scientific Visualization and Computer Graphics. In *IEEE Visualization Course Notes*, 2002.

[25] P. Sulatycke and K. Ghose. A fast multithreaded out-of-core visualization technique. In *International Parallel Processing Symposium / Symposium on Parallel and Distributed Processing*, 1999.

[26] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, 1997.

[27] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale. DI-MMAP–a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 2013.

[28] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale. On the role of NVRAM in data intensive HPC architectures: an evaluation. In *IEEE International Parallel & Distributed Processing Symposium*, 2012.

[29] M. S. Ware, K. Rajamani, M. S. Floyd, B. Brock, J. C. Rubio, F. L. R. III, and J. B. Carter. Architecting for power management: The IBM POWER7(TM) approach. In *16th International Conference on High-Performance Computer Architecture*, pages 1–11, 2010.

[30] H. Yu, C. Wang, and K.-L. Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. In *International Conference on High Performance Computing Networking, Storage and Analysis*, pages 24:1–24:12, 2007.