Flow-Guided File Layout for Out-Of-Core Pathline Computation

Chun-Ming Chen

Boonthanome Nouanesengsy

Teng-Yok Lee

Han-Wei Shen

The Ohio State University *

ABSTRACT

As CPU processing power becomes more powerful and storage capacity increases, performing data-intensive visualization computations involving large data on a desktop computer becomes an increasingly viable option. Desktops, though, usually lack the memory capacity required to load such large data at once, and thus the cost of I/O becomes a major bottleneck for the necessary out-of-core computation. Among techniques that reduce runtime I/O cost, reordering the file layout to increase data locality has become popular in recent years. However, file layout techniques for time-varying scientific data, especially for time-varying flow fields, have been rarely discussed. In this paper, we evaluate the performance impact of utilizing a file layout method for out-of-core time-varying flow visualization. We extend a graph-based representation of flow fields, originally developed for static vector fields, to time-varying flow fields, and apply a graph layout algorithm to order data blocks to be written to disk. Benefits from the generated file layouts are evaluated using various parameters and seeding scenarios.

1 INTRODUCTION

The rapid growth of data size in recent years has made scientific visualization more challenging. Although very large-scale data analysis and visualization tend to be performed on supercomputers, scientists still prefer to analyze data on their local machines whenever possible. With more powerful multi-core CPUs and increasing storage capacities, a higher-end desktop computer can now handle much larger data size than before. Desktops, though, usually have insufficient memory capacity to hold the entire data at once. Therefore, it is necessary to perform visualization of large data in an out-of-core manner, that is, data are brought from disk to main memory on demand. Out-of-core computation suffers from unavoidable slow disk I/O in terms of low bandwidth and high latency, thus reducing the I/O overhead still remains an essential issue.

Time-varying flow fields comprised of vector fields with many timesteps are a typical case of large data. To visualize time-varying flow fields, generating pathlines is a popular method, since it is also widely used as building blocks for other visualization techniques, e.g. streaklines and path surfaces. Conventionally, time-varying flow data are stored in separate files for each timestep. However, this default file layout is disadvantageous for out-of-core pathline computation because pathlines will require data from at least two timesteps at each integration step, which will lead to frequent I/O requests in many different disk locations. We tackle the problem by allowing data in different timesteps to be stored together in the same file, and reorder the file layout to further increase data locality.

To reduce the I/O overhead for out-of-core computations, reordering the file layout for better data locality has become popular in recent years. However, few file layout methods for efficient out-of-core pathline computation have been discussed. In this work, we first divide the flow field in each timestep into spatial blocks, and blocks in the same spatial location but in sequential timesteps

are merged together to form a *time block*. We employ small block sizes, as suggested by Cox and Ellsworth [5], in conjunction with a runtime file reader that prefetches several contiguous time blocks at each disk access. In order to maximize the usage of this prefetched data, we generate a storage order that prioritizes placing time blocks which are to be consecutively used at runtime together in the file layout.

To compute a storage order of time blocks that follows the runtime access pattern during pathline computation, we extend the flow-guided file layout algorithm that was originally designed for out-of-core streamline computation, proposed by Chen *et al.* [3], to time-varying data. We observe that the runtime access pattern among time blocks follows the flow direction, and can be predicted during a preprocessing stage, in which a graph representation of the flow field is generated. The graph nodes represent a time block, and a graph edge connects two nodes if there exist particles traveling between them. Using this graph to model the runtime access pattern, a layout can be computed by minimizing the sum of distances between each pair of dependent blocks.

The contributions of this paper are that we extend a previous flow-guided file layout algorithm designed for static flow fields to time-varying flow fields, and evaluate the benefits from the generated layout using various parameter values and seeding scenarios. We show the I/O performance gain from using our layouts compared to a general space-filling-curve layout.

2 RELATED WORK

Of out-of-core techniques, Silva *et al.* [10] provides a review in scientific visualization. Cox and Ellsworth [5] proposed to store data in spatial blocks to improve data locality and used application controlled demand paging for better I/O performance. Ueng *et al.* [12] presented a technique to compute streamlines in large unstructured grids with a queuing system. Pugmire *et al.* [9] used a hybrid of static block allocation and load-on-demand to achieve load balance and minimal I/O. Nouanesengsy *et al.* [7] used flow graph to predict and balance the runtime workload for parallel streamline computation. Camp *et al.* [2] cached flow data in solid state drives to reduce overhead of repeated I/O. For visualizing time-varying flow fields out-of-core, Bruckschen *et al.* [1] took a different approach, which computes and stores pathlines in a pre-processing stage.

Reordering layouts for visualizing multi-dimensional data out-of-core has also been an active area of research. Sulatycke and Ghose [11] proposed a multi-threaded system for out-of-core isosurface rendering with a layout that reduced disk seeks. Chiang [4] used a cache-oblivious time tree for isosurface extraction of irregular time-varying data. Yoon and Lindstrom [13] proposed a cache-oblivious layout algorithm for visualizing meshes.

3 LAYOUT ALGORITHM AND OUT-OF-CORE SYSTEM

In our framework, we add a preprocessing stage before pathlines are computed. In this stage the flow field is analyzed and modeled as a graph. Using this graph, the file layout is then optimized. Since preprocessing can take significant time, this stage should be done only once for each flow dataset. Therefore, the layout algorithm is designed to accommodate various disk parameters, so the reordered file can be reused on different machines with different seed locations.

*e-mail: {chenchu,nouanese,leeten,hwshen}@cse.ohio-state.edu

Figure 1: The construction of Access Dependency Graph (ADG). (a): An illustration of a one-dimensional time-varying flow field with five timesteps and four time blocks. Time blocks 1 and 2 contain data in timesteps 1 to 3. Time blocks 3 and 4 contain data in timesteps 3 to 5. The grey dotted arrows show the underlying flow. Each time block has four sampling seeds. (b): The ADG of one hop constructed from (a). (c): The ADG of two hops. (d): The union of (b) and (c).

3.1 Graph Generation and Layout Algorithm

The goal of our file layout is to provide an ordering of the data blocks which benefits prefetching and hence increases I/O efficiency at runtime. In other words, the layout order should follow the runtime access pattern during the pathline computation. In our approach, the runtime access pattern is predicted and modeled using a graph representation of the flow field. First, each timestep in the time-varying flow field is divided into equal sized spatial blocks, and then blocks in the same spatial location but in sequential timesteps are merged together to form a time block. In order to store time blocks in a file layout with higher data locality, we extend the layout method proposed by Chen *et al.* [3], originally designed for streamline computation. The main difference between the previous work and the algorithm in this paper is that the graph construction procedure is applied to time-varying flow fields. We model the runtime access pattern of pathline computation as a directed graph called *access dependency graph*, or ADG. In the ADG, each graph node represents a time block of the dynamic flow field. A directed graph edge connects two time blocks if there exist pathlines passing through them. Each edge is assigned a weight equal to the conditional probability of seeds moving from one block to the other. The generation of the ADG is shown in Figure 1(a). Sampling seeds are densely placed in each time block, and are then traced throughout the flow field. The edge weights are calculated by finding the percentage of particles which, starting from each time block, travels to its immediate neighboring block. Here a neighboring block can be the time block in the same spatial location but with different extents in the time dimension, as a particle moving in time can hit a boundary in either the spatial or temporal domain. The constructed ADG now encapsulates the dependencies of all immediately connected time blocks, as illustrated in Figure 1(b).

An ADG that represents the connectivity between neighboring blocks is not accurate enough because it fails to capture the potentially higher dependencies between blocks which are more than one block away. The term *hop* is defined as the number of blocks a particle has traveled through from its initial location, excluding the block where the particle begins. The previously discussed ADG is defined as a 1-hop ADG. An N -hop ADG is constructed by allowing sampling particles to advect through N blocks from their starting block. After all the sampling particles have traveled through N blocks, the number of particles in each ending block is counted and normalized as the edge weight. For simplicity, an N -hop ADG is denoted by G_N . The G_2 graph is shown in Figure 1(c), where all the four sampling particles in time block 1 travel to time block 4. As can be seen, edge (1,4) in G_2 has higher weight than any edge in G_1 , and cannot be predicted from G_1 . Since the goal of the layout is to place highly dependent blocks together, we cannot ignore the potentially higher edge weights in the multi-hop ADG. We use the union of graphs G_1 to G_N , denoted by G_N^* as the final ADG.

After the ADG model is constructed, the optimal layout problem can be formulated as a graph linear arrangement problem. The goal

Figure 2: Parallel out-of-core visualization system.

is to minimize the sum of distances in the file layout for each pair of connected nodes. We use a cost function that takes the generated ADG and a layout as the input, and returns the total cost, which is the sum of the costs from all the edges in the ADG. Each edge cost is proportional to the edge weight and the distance of the two connected nodes in the layout. Since most graph linear arrangement problems are NP-hard [6], we use the approximation algorithm proposed by Chen *et al.* [3], which recursively divides the graph into two subgraphs with the minimum edge-cut and optimizes the layout for each subgraph individually in a divide-and-conquer manner.

3.2 Out-of-core Parallel Pathline Computation System

To compute pathlines out-of-core, we use a multi-threaded system to overlap I/O time with computation time. In the system there are two types of threads, an I/O thread and several pathline computation threads. Each type is associated with a queue to manage I/O or computation requests, as illustrated in Figure 2. The I/O thread also manages a memory pool which caches prefetched or previously used data blocks. At the beginning of the program, the initial seeds are divided into separate tasks based on their block region. All tasks are then placed into the computation queue. The computation threads remove tasks from the computation queue when they require more work, and check whether the data block required to advect particles in the task are ready in the memory pool. If not, an I/O request is placed into the I/O queue, and the particles in the task are temporarily suspended.

The I/O thread loads data blocks from disk based on the requests in the I/O queue. If the memory pool is full, currently unused blocks are released in least recently used manner. When the I/O is complete, the computation request is then pushed back to the computation queue and the previously suspended particles can proceed. In order to reduce I/O accesses while utilizing the generated file layout, the data reader uses prefetching to reduce the number of I/O accesses. Given a prefetch size, the prefetching loads consecutive data blocks ahead of the requested block in a single disk access. With an optimized file layout, the prefetched data cached in the memory pool will be used in the near future and hence will increase the hit rate of the cache.

4 RESULTS

Three time-varying datasets were used in our experiments: *Isabel*, *Plume*, and *Smhagos*. The *Isabel* dataset simulates hurricane Isabel from September 2003 over the west Atlantic region. The *Plume* dataset is a simulation of the thermal downflow plumes on the surface layer of the sun. The *Smhagos* dataset is a climate simulation over the Indian and Pacific Ocean. The size of each dataset is listed in Table 1.

We compared the performances of our layouts optimized for ADG G_1 , G_3^* and G_5^* , as well use as the Z-order space-filling curve (Z-curve) as a baseline. The Z-curve layout, which is widely used for multi-dimensional data [8], traverses the space in different dimensions at each recursion level. We applied the finest traversing step in the time dimension in the forward direction.

Since we utilize prefetching to reduce I/O requests, we divided the flow field into smaller-sized blocks and apply a layout to increase their spatial locality. The spatial block size used in our tests

Table 1: Dataset properties, ADG statistics and preprocessing time. The four numbers within columns 5 and 7 represent for ADG $G_1/G_3^*/G_5^*$

Dataset	Dimension	File Size	# Blocks	# Edges (millions)	ADG Generation	Layout Generation	Reordering
<i>Isabel</i>	$500^2 \times 100 \times 48$	14.4 GB	336,896	2.6/7.1/12.3	8/28/45 min	6/7/8 min	15 min
<i>Plume</i>	$252^2 \times 1024 \times 29$	22.6 GB	458,752	3.0/9.5/16.5	16/58/104 min	12/14/16 min	21 min
<i>Smhagos</i>	$2699 \times 599 \times 50 \times 25$	24.3 GB	616,512	3.1/11.3/22.0	17/74/145 min	23/25/26 min	43 min

Figure 3: Layout validation through the miss rate of sequential particle tracing. The curves represent the Z-curve (....) and the flow-guided layouts optimized for G_1 ($\triangle$), G_3^* (∇), and G_5^* ($\circ$).

Figure 4: The miss count (color bars) and the total number of loaded blocks (white bars) for uneven seeding. Note that the box counts are plotted in logarithm scale. The bars in each group from left to right represent the result from the Z-curve layout and our layouts optimized for G_1 , G_3^* and G_5^* .

Figure 5: The block loading time (color bars) and wall time (white bars) of uneven seeding. The bars in each group from left to right represent the result from the Z-curve layout and our layouts optimized for G_1 , G_3^* and G_5^* .

was 16^3 . Every two contiguous timesteps were grouped to form a time block. In other words, a time block may span timesteps one and two, two and three, and so on.

4.1 Preprocessing

Our flow-guided file layout requires data preprocessing to construct the ADG and to compute the layout. In our experiments, each ADG edge weight is computed by placing one particle per 2^3 voxels at each timestep. We used the out-of-core system described in Section 3.2 to trace particles and construct the ADG. The preprocessing was done on a Linux workstation with an Intel Xeon CPU, 24 GB of RAM, and a RAID-5 disk array. Table 1 shows the preprocessing time, including graph construction, layout generation and file reordering. To speed up, each ADG was generated using eight computation threads.

4.2 Sequential Particle Tracing

The I/O performance of the out-of-core pathline computation can be affected by both the design of the out-of-core system and the layout for the flow field. To verify the effectiveness of the file layout using a prefetching system, we randomly placed seeds in the domain and traced one particle at a time using various prefetch sizes. The prefetch size represents how many blocks ahead of the requested block are to be loaded into the memory in a single I/O call. Given a prefetch size p , the data loader reads $p + 1$ blocks contiguously, including the requested block. Tracing one particle at a time ensures that the resident blocks in the memory pool are not prefetched by other seeds, which affects the accuracy of the miss rate. During sequential particle tracing, the number of disk accesses was counted, and in the end the miss rate was calculated by dividing the total number of disk accesses to load missing blocks by the number of blocks the seed has traversed. If the layout is good, more prefetched blocks will be used and hence the cache misses are reduced.

The results for the three datasets are shown in Figure 3. As can be seen, our layouts using three different numbers of hops always achieve lower miss rates than the Z-curve layout. Among our layouts, the ones optimized for G_3^* and G_5^* achieves a lower miss rate than that using only G_1 when the prefetch size is larger than two. This is because the ADG with higher number of hops contains more information about the runtime access pattern. Within the tested datasets, the curves for *Smhagos* show higher miss rates than those in the other datasets.

4.3 Runtime Performance

We tested the layout performances for out-of-core pathline computation based on different seeding scenarios. The first was *uneven seeding*, where particles were seeded in a small region, to visualize the flow starting from a specific region. The other was *uniform seeding*, where particles were evenly distributed in the domain, which provides an overview of the flow field. For each seeding scenario, small and large seed sizes were tested. The experiments were run on a Linux machine with a 7200 RPM disk and 4 GB of RAM. Four computation threads were used with the Intel Core i7 - 2600 CPU. In all tests, we limited the memory pool size to be 2 GB, which was able to hold around 14,000 time blocks.

Uneven Seeding In this test particles were seeded in a 32^3 region at the first timestep. Several randomly selected regions were tested for each layout. During pathline computation, the miss count and the actual number of blocks loaded due to prefetching are counted, as shown in Figure 4. The miss count corresponds to the total number of I/O accesses due to a memory cache miss. A reduced miss count indicates better I/O efficiency. In the results using small seed size, we can see that the miss count of our layouts are all less than that of the Z-curve layout (blue bars) when the prefetch size is larger than one. Also, for each prefetch size, the miss count trends downward as the layout uses more hops. For large seed size, the same trend can be seen, with layouts which use more hops having the lowest miss count, except for the *Plume* dataset with similar miss counts.

Figure 5 shows the accumulated I/O time (color bars) and total running time (white bars). All the graphs show that the total running time is very close to the I/O time, indicating that the I/O cost dominates the out-of-core pathline computation. From all test cases, we can see that when the prefetch size is larger than one, our layout achieves better performance versus the Z-curve layout. In most

Figure 6: The miss count (color bars) and the total number of loaded blocks (white bars) for uniform seeding. Note that the box counts are plotted in logarithm scale. The bars in each group from left to right represent the result from the Z-curve layout and our layouts optimized for G_1 , G_3^* and G_5^* .

Figure 7: The block loading time (color bars) and wall time (white bars) of uniform seeding. The bars in each group from left to right represent the result from the Z-curve layout and our layouts optimized for G_1 , G_3^* and G_5^* .

cases, better performance is gained from layouts which use a higher number of hops for the ADG. By examining each layout individually, we can see that as the prefetch size increases, the computation time first decreases, and then goes up again. This is because with an increased prefetch size, the miss rate should decrease, and the number of disk accesses is also reduced. However, when the prefetch size is too high, the miss rate cannot be further reduced, thus unnecessary disk reads will increase I/O time. Since our layout for *Smhagos* generally has a higher miss rate than our other datasets, as shown in Figure 3, the best prefetch size also becomes lower. From the results we can empirically determine that the best layout is our layout using G_5^* , and the proper prefetch size is within 3 and 7.

Uniform Seeding In this test particles were evenly distributed in the domain. Unlike the previous seeding scenario where most particles share similar starting paths and mostly pass through the same blocks, this test is more I/O intensive because each seed will request data blocks in disparate regions of the flow field. Figure 6 shows the miss counts and the total number of loaded blocks, and Figure 7 shows the I/O and total running time. From the results we see trends similar to the uneven seeding results. When the prefetch size is larger than one, our layouts achieve higher performance than the Z-curve layout, and a decreasing runtime can be seen when using higher number of hops, except for using the *Smhagos* dataset or the 4096 seed set size. From the results, the layout optimized for G_3^*

Table 2: The percentage of the runtime reduction of our layout using G_3^* over the Z-curve layout with prefetch size 7.

Dataset	Uneven seeding		Uniform seeding	
	64 seeds	4096 seeds	64 seeds	4096 seeds
<i>Isabel</i>	41%	38%	39%	25%
<i>Plume</i>	21%	10%	29%	31%
<i>Smhagos</i>	24%	19%	31%	36%

achieves the best total time in most cases, and the proper prefetch size is still within 3 and 7.

Table 2 lists the percentage of runtime reduction when using the layout computed from G_3^* , compared to the running time of the Z-curve layout, both under the prefetch size 7. As shown in the table, at most 41% running time can be reduced.

5 CONCLUSION

In conclusion, we have presented a data layout method and a runtime prefetching scheme for out-of-core pathline computation. We compared our layout with the Z-curve layout in different seeding scenarios using various parameters, and showed that using the knowledge of the access pattern encoded in ADGs, our layout can achieve higher I/O efficiency. Since ADG construction is the bottleneck in our layout method, a future work is to improve the performance as well as the accuracy of the ADG.

ACKNOWLEDGEMENTS

This work was supported in part by NSF grant IIS-1017635, US Department of Energy DOE-SC0005036, Battelle Contract No. 137365, and Department of Energy SciDAC grant DE-FC02-06ER25779, program manager Lucy Nowell.

REFERENCES

- [1] R. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-time out-of-core visualization of particle traces. In *Proceedings of the IEEE symposium on parallel and large-data visualization and graphics*, pages 45–50, 2001.
- [2] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. Joy. Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *LDAV '11: IEEE Symposium on Large Data Analysis and Visualization*, pages 57–64, 2011.
- [3] C.-M. Chen, L. Xu, T.-Y. Lee, and H.-W. Shen. A flow-guided file layout for out-of-core streamline computation. In *PacificVis '12: IEEE Pacific Visualization Symposium*, pages 145–152, 2012.
- [4] Y.-J. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Vis '03: IEEE Visualization*, pages 217–224, 2003.
- [5] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Vis '97: Proceedings of IEEE Visualization and Visualization*, pages 235–244, 1997.
- [6] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.
- [7] B. Nounesengsy, 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.
- [8] V. Pascucci and R. Frank. Global static indexing for real-time exploration of very large regular grids. In *SC '01: ACM/IEEE Supercomputing*, page 45, 2001.
- [9] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *SC '09: ACM/IEEE Supercomputing*, pages 16:1–16:12, 2009.
- [10] C. T. Silva, Y.-J. Chiang, J. El-sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. *Visualization 02*, 2002. Course Notes, Tutorial 4.
- [11] P. Sulatycke and K. Ghose. A fast multithreaded out-of-core visualization technique. In *PDP '99: Parallel and Distributed Processing*, pages 569–575, 1999.
- [12] 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.
- [13] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, 2006.