Flow-guided file layout for out-of-core pathline computation

Article · October 2012
DOI: 10.1109/LDAV.2011.6092326

4 authors, including:

Teng-Yok Lee
Mitsubishi Electric Research Laboratories
34 PUBLICATIONS **441** CITATIONS

SEE PROFILE

A Flow-Guided File Layout for Out-Of-Core Streamline Computation

Chun-Ming Chen Lijie Xu Teng-Yok Lee Han-Wei Shen

The Ohio State University *

Abstract

We present a file layout algorithm for flow fields to improve runtime I/O efficiency for out-of-core streamline computation. Because of the increasing discrepancy between the speed of processors and storage devices, the cost of I/O becomes a major bottleneck for out-of-core computation. To reduce the I/O cost, loading data with better spatial locality has proved to be effective. It is also known that sequential file access is more efficient. To facilitate efficient streamline computation, we propose to reorganize the data blocks in a file following the data access pattern so that more efficient I/O and effective prefetching can be accomplished. To achieve the goal, we divide the domain into small spatial blocks and order the blocks into a linear layout based on the underlying flow directions. The ordering is done using a weighted directed graph model which can be formulated as a linear graph arrangement problem. Our goal is to arrange the file in a way consistent with the data access pattern during streamline computation. This allows us to prefetch a contiguous segment of data at a time from disk and minimize the memory cache miss rate. We use a recursive partitioning method to approximate the optimal layout. Our experimental results show that the resulting file layout reduces I/O cost and hence enables more efficient out-of-core streamline computation.

1 Introduction

As the processing power and storage capacity of supercomputers continue to increase, the size of data generated from large scale simulations also grow drastically. Although there is a trend to move computation for data analysis and visualization to supercomputers as much as possible, most scientists still prefer to analyze data in their offices using desktop computers whenever possible. Nowadays, multi-core desktop computers capable of small scale parallelism have become pervasive. Machines of this type, however, usually have a much smaller amount of memory compared to the size of data. To cope with the limitation, visualization of large data sets needs to be done in an out-of-core manner, that is, data are brought from disk to main memory on demand. Because the improvement in the I/O devices both in terms of latency and bandwidth has failed to keep up with the speed increase of processors, the bottleneck of large scale data analysis and visualization has now shifted to I/O.

For vector field data, displaying streamlines is still the most popular visualization method. Efficient out-of-core computation of streamlines, however, is not straightforward because the data access pattern for computing streamlines is not always I/O friendly. On the one hand, to improve spatial locality and thus increase the utilization of data after they are brought into main memory, the size of data blocks should be kept as small as possible, as suggested by Cox and Ellsworth [4]. On the other hand, loading a large number of smaller blocks will be more costly if those blocks are scattered in disk, incurring longer disk seek time and a larger number of small I/O requests.

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

It is known that prefetching data blocks can be an effective way to bridge the widening gap between disk access time and processor speed. To maximize the benefit of prefetching, data blocks should be placed in disk in the order that conforms with the application's data access pattern. Taking streamline computation as an example, to ensure that the data blocks ahead of the current particle position will be readily available, one can organize the data blocks in disk according to the flow directions, and then prefetch contiguous chunks of data at run time whenever possible. To facilitate efficient out-of-core streamline computation, in this paper we present a file layout algorithm for flow data. Our goal is to order the data blocks in such a way that the I/O cost be minimized and the utilization of prefetch data maximized for streamlines seeded at arbitrary locations in the field.

To achieve our goals, we introduce a directed graph model, called the *access dependency graph* or ADG, to model the data access pattern. The graph nodes represent spatial data blocks, and an edge connects two blocks if there exist particles traveling between them. The ADG considers not only spatially adjacent blocks, but also non-adjacent blocks if they are on the same flow path of a particle. Using ADGs, the file layout problem can be formulated as a linear graph layout problem where the sum of distances in the file for data blocks along the same flow paths is minimized.

As recomputing the file layout for different hardware configuration is highly undesired, we want the file layout to always incur smaller I/O overhead even for machines of different memory capacities, a common goal shared by most of the cache-oblivious mesh layout algorithms [1, 7, 17]. Compared to these algorithms, which mainly consider undirected connectivities among adjacent mesh cells, our layout algorithm is designed for directed graphs by taking flow directions into account, leading to a higher utilization of the prefetched blocks for efficient streamline computation.

The major contributions of this paper are that we propose a graph-based model as a succinct representation of flow fields that are too large to store in main memory. By using this graph model, we can more easily estimate the rate of utilization for prefetched data blocks and in turn design an optimization algorithm to generate an optimal file layout. Based on the optimized layout, we developed an efficient out-of-core visualization system to generate streamlines for large scale data sets.

The rest of the paper is organized as follows. In Section 2, we review the previous works. After the overview of our system in Section 3, Section 4 discusses the graph model and the layout optimization algorithm. Section 5 describes the implementation details for the system's data loader and visualization components. In Section 6, we discuss the experimental results and compare our method with other conventional layout algorithms. Finally, we address the limitations in Section 7 and conclude this paper in Section 8.

2 Related Work

Designing efficient out-of-core algorithms has been an active area in visualization research. Silva *et al.* give a comprehensive review of out-of-core visualization techniques [12]. For flow visualization, Cox and Ellsworth [4] propose to store data in 3D blocks to improve data locality and to use application-controlled demand paging over system paging for better I/O performance. Sulatycke and Ghose [13] propose a multi-threaded out-of-core system to reduce

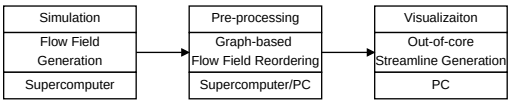

Figure 1: System Overview.

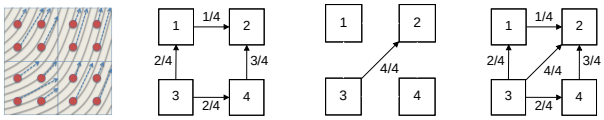

Figure 2: The Access Dependency Graph (ADG). (a): An illustrative flow field divided into four blocks. The flows are aligned to the grey curves with the direction from bottom left to top right. (b): The ADG of one hop. (c): The ADG of two hops. (d): The merge of (b) and (c).

IO overhead. Ueng *et al.* [15] propose a streamline computation system for irregular grid by organizing data using an octree data structure. Bruckschen *et al.* propose to compute streamline traces in a preprocessing stage and load the precomputed streamline traces on demand [2]. Our work is related to the out-of-core system on [4] and the multi-thread system by [13], with a special focus on prefetching and reducing I/O cost by file reordering.

Re-arranging file layouts to improve I/O or cache performance has received much research attention in the past decade. Pascucci and Frank implement a global static indexing scheme using Z curves for real time slicing of very large scaled regular grids volumetric data [11]. Niedermeier and Sanders use Hilbert curve [5] to create efficient indexing for mesh-connected computers for parallel computing [9]. Isenburg and Lindstrom propose an efficient streaming mesh format that is suitable for effective disk I/O for out-of-core applications [6]. Researchers have also proposed schemes to improve cache efficiency by reorganizing large-scaled mesh data [1, 7, 17]. In their methods, the mesh is formulated as a graph and the data reorganizing problem is reduced to the graph arrangement problems. Among the above works, Yoon *et al.* propose a novel cache-oblivious cost function using the geometric mean of possible cache sizes, which can generate better layouts than using arithmetic means [17]. Tchiboukdjian *et al.* [14] propose a layout algorithm for unstructured meshes with a theoretic complexity analysis that can be represented by overlapping graphs.

Recently, modeling flow fields as graphs starts to become popular. The advantage of using graph models is that it provides a compact representation for large scale flow fields. Chen *et al.* [3] use the *Morse Connection Graphs* (MCG) model to represent the topological structure of the flow fields. The MCG model facilitates flow field analysis such as simplification, periodic orbits generation and extraction. Xu and Shen [16] propose a different node link graph model used mainly as user interface. A similar model is applied for load-balanced parallel computation of streamlines by Nouanesengsy *et al.* [10]. Based on the graph model and the initial seeds, the computation workload needed for each block can be estimated, which is then used to optimize data partitioning and balance workload.

3 System Overview

This section provides an overview of our system, where the goal is to compute and render streamlines for large scale flow fields in an out-of-core manner. Figure 1 shows the key components in our system. Because the data set generated from large scale simulations running on a supercomputer is often too large to fit into the PC's main memory, a preprocessing stage is necessary to decompose the data into smaller blocks and store them back to a file. Decomposing the entire flow field into smaller blocks can reduce the working set size as well as improve the data's spatial locality if the application's access pattern is taken into account.

To minimize the I/O overhead, our out-of-core system loads data blocks from disk only when they are needed. Clearly, when performing streamline computation, the only block that is absolutely required is the block that contains the particle's current position. To allow the particle to move forward for a longer distance without having to frequently stop and request more data, we can load more data blocks at the down stream of the flow from the current particle position, in other word, to prefetch the data that will be needed soon. Since sequential access to disk is more efficient than random access, the blocks that will be prefetched should be placed in a close proximity in the file.

Our system automatically determines the layout of the data blocks in a file to optimize the efficiency of data prefetching when computing streamlines. As the locations of the particles are unknown in the preprocessing stage, we represent the access dependency among the data blocks in a flow field as an graph, where each node represents a spatial data block, and each edge represents the probability for the particles to travel from one block to the other. We then linearize the graph nodes and use the order to place the corresponding data blocks in a file. In the following, we explain our algorithm used in the preprocessing stage in more detail.

4 Graph-based Flow Field Data Layout

In this section we present a graph-based algorithm to lay out flow field data. We first introduce a graph model that encodes the flow directions in the field. The graph will provide important hints for us to estimate the I/O cost for any given layout of a flow field file. With the graph and the I/O cost model, we can compute an efficient file layout by solving an optimization problem. Our algorithm allows efficient prefetch of data blocks when computing streamlines in an out-of-core manner by placing data blocks that will be fetched together in a close proximity in a file. This not only reduces the disk seek time, but also helps the data utilization rate in the memory cache.

4.1 The Access Dependency Graph (ADG)

4.1.1 Flow Graph Model and Construction

Here we introduce the graph model, called *Access Dependency Graph* (ADG) to model the dependency of data access among data blocks. The ADG model is a generalization of the graph models presented in [16] and [10]. Given a flow field, we create a directed graph from it where each node represents a spatial data block, and each directed edge between two nodes represents the flow between the two blocks. The edge weight records the probability for a particle uniformly seeded in the block to travel to the other block. To estimate the weight, we distribute particles randomly within each block, and then compute the percentage of the particles moving from one block to the other blocks represented by the graph nodes.

Figure 2 gives an example of ADG, where the vector field is divided into 2×2 blocks, shown in Figure 2(a). To compute the edge weights, we uniformly place seeds within each block and then calculate the percentages of the seeds traveling to its neighboring blocks, shown in the corresponding ADG in Figure 2(b).

The construction of ADG depends on two parameters. The first one is the block size, which affects the memory footprint size needed for the flow graph, and the accuracy of the graph model. In our implementation, we empirically chose the block size as 16^3, which will be explained in more detail in Sec. 6.

The second parameter is the distance that the particles are allowed to travel, which will affect the accuracy when we try to predict the needed data blocks when a particle travels multiple *Hops*. The control of particle hops differentiates our work from the graph models used by Xu and Shen [16] and Nouanesengsy *et al.* [10].

4.1.2 1-Hop vs *N*-Hop Flow Graph

Hereafter we refer to *hop* as the number of blocks that a particle passes through as it advects in the domain, excluding the block that the particle is originated. For example, a particle moves *one hop* if it advects from its originating block to the next block, and *two hops* if it continues to move to the third block. A *1-hop* graph, denoted as G_1, is a graph constructed by only allowing the uniformly sampled particles to move one hop. In other words, graph links are created only between a block and its immediate neighbors in the spatial domain.

To generate an *N-hop* graph, we advect the particles from each block, i.e. a graph node, long enough until they travel N blocks away from their starting block. We count the number of particles in each of the blocks those particles have visited, normalize the counts, and then use the normalized values as the edge weights between the originating graph node and the corresponding nodes of those N blocks. Each edge weight represents the percentage of streamline seeds moving to each of the connected nodes. We call this generalized flow graph as a *N-hop* graph, denoted as G_N. Figure 2(c) shows a graph of 2 hops for the same flow field.

The reasons for us to create *N-hop* graphs is that the G_1 graph alone cannot represent the flow field well, which will in turn affect the quality of the resulting file layout. Even though mathematically speaking, for example, the 2-hop graph can be predicted from the 1-hop graph by using the Markov chain since the edge weights represent the conditional probability between two nodes, the prediction will involve errors. As the example shown in figure 2(c), the actual probability seeds moving from node 3 to node 2 in two hops is 100% , where the prediction from G_1 is only 50% if we sum up the probabilities of two paths through node 1 and node 4. This error will become even higher if we further estimate G_3 from G_1.

4.2 Layout Cost Function

The flow directions encoded in the ADG provides us the necessary information to deduce the data access pattern when computing streamlines. We can also use it to estimate the I/O cost for a layout where the data are already decomposed into blocks and placed in a file.

As previously described, the goal to reorder the data blocks of a flow field is to place consecutive blocks along the same particle trace in a close proximity in a file such that they can be prefetched together from disk to main memory with a minimum I/O overhead, i.e, fewer disk reads. It also allows us to increase the utilization of the prefetched blocks, or equivalently, to reduce the miss rates when the required data blocks are not in the prefetched block pool. In this section, we explain how to estimate the I/O cost for a layout L. In essence, a layout L is a one-to-one mapping from a node n in ADG to a location $L(n)$ in the file. For example, L(4)=2 indicates that node 4 is placed in the second block of the file from the beginning. Below we first introduce the cost function for a one-hop graph and then extend the cost function to multi-hop graphs.

4.2.1 Single-hop Graph-Based Cost Function

To reduce the I/O overhead for computing streamlines out-of-core, when a block b is requested to be loaded, our system also prefetches several additional blocks in the file immediately following the requested block. Depending on how the data blocks are laid out in a file, the block that will be needed by the current particle right after b will be successfully prefetched if the block is placed in the file at a distance within the prefetched data size from b and along the forward direction. In other words, the next required block will not be prefetched if the block is placed in the back of b or the distance in the file between the two blocks is greater than the prefetch data size. Hence a memory cache (the prefetch pool) miss occurs. Based on this idea, we can estimate the miss rate for data blocks after prefetching by examining the distance of every possible pair of blocks in the file that are to be accessed contiguously during particle tracing.

Formally, given a layout L and the access dependency graph G_1, we express the cost of the layout using the *Estimated Miss Ratio* (EMR), as shown in Equation 1. In essence, it is the averaged prefetch miss rates for all pairs of directly connected graph nodes.

$$EMR(L,G_1) = \frac{1}{|V|} \sum_{(u,v)\in G_1} Pr_1(v|u) \times PMRF(L(v)-L(u)) \quad (1)$$

Here $|V|$ is the number of nodes in G_1, and $Pr_1(v|u)$ is the conditional probability of node v to be directly accessed from node u. This is the weight for the edge (u,v) in G_1 estimated by using the method mentioned above.

The term *Prefetch Miss Ratio Function* (*PMRF*) is a probability function that returns the probability of a miss given the layout distance between two blocks to be accessed sequentially. If we know the exact prefetch size, say B, we can define $PMRF^B$ as in Equation 2, which is essentially checking whether the distance between two adjacent blocks in the file is within the prefetch size B:

$$PMRF^B(d) = \begin{cases} 0 & : 0 \le d < B \\ 1 & : otherwise \end{cases} \quad (2)$$

The block distance d is an input to the function $PMRF()$, which can be negative, meaning the next block to be accessed is located *behind* the current block in the file.

In practice the value of B is a tunable parameter defined by the visualization system. Since it is often not known at the stage the file layout is being designed, we use Equation 3 to model the prefetch size:

$$PMRF^O(d) = \begin{cases} \frac{lg(d+1)}{lg|V|} & : d \ge 0 \\ 1 & : d < 0 \end{cases} \quad (3)$$

which is a monotonically increasing function for positive offsets. The value of this function quickly approaches to 1 for a moderate size of d. This is because when the offset d is too large, the miss rate should be one even if the prefetching size is unknown. It is noteworthy that the positive portion of this cost metric is used by Yoon and Linderstorm as a geometric cache-oblivious metric [17].

4.2.2 Multi-Hop Graph-Based Cost Function

The cost function based on one-hop graphs only examines the prefetch miss rate between two sequentially accessed blocks. As described in section 4.1.2, since the longer access behavior is not well-modeled by the one-hop graph, the evaluated cost may not reflect the real I/O cost in longer hops. To model the I/O cost more precisely, we need to use the multi-hop graphs including $G_1, G_2, \ldots G_N$ as well. Essentially, our goal is to model the conditional probabilities of all blocks to be accessed in the next N hops, given the initial block. We use the graphs $G_1, G_2, \ldots G_N$ to calculate the probability of each block that can benefit from prefetching. Equivalently, we estimate the prefetch miss ratio within N hops in Equation 4:

$$EMR(L,\{G_1,\ldots,G_N\}) = \quad (4)$$
$$\frac{1}{|V|N} \sum_{i=1}^{N} \sum_{(u,v)\in G_i} Pr_i(v|u) \times PMRF(L(v)-L(u))$$

where $Pr_k(v|u)$ is the conditional probability of node v to be accessed from node u in k hops, which is equal to the weight of edge (u,v) in G_k.

One issue to address in our cost model is the choice of N. In practice, as N grows, the particle will scatter into more blocks, and hence each individual edge weight in G_N will become smaller. This results in smaller total weights in the accumulated graph. In fact, we have observed that N does not need to be large in order to get a satisfactory prefetching result. More details about the choice of N will be discussed in Section 6.

4.3 The Layout Algorithm

After we define the cost of a layout due to memory cache miss in Equation 4, we can start to search for the file layout with a minimal cost. From the cost equation, it appears that the calculation of the cost involves N graphs, but we can rewrite the cost function as below in Equation 5:

$$EMR(L, G_N^*) = \frac{1}{|V|N} \sum_{(u,v)\in G_N^*} w(u,v) \times PMRF(L(v) - L(u)) \quad (5)$$

where $G_N^* = \{G_1, G_2, ..., G_N\}$. An example of G_N^* is shown in Figure 2 (d) where $N = 2$. Essentially, the N graphs are merged into a single graph by taking the edges from each graph into a single multi-hop ADG. After we can estimate the miss ratio of the multi-hop ADG for a given layout, the next step is to find the optimal layout that minimizes the miss ratio in Equation 5. This optimization problem is similar to the family of graph linear assignment problems, which are NP-hard [8]. To find an approximated solution, our implementation adopts the layout algorithm used in [1, 7, 17]. Given an undirected graph, this algorithm reduces the problem size by recursively cut the graph into two partitions of similar size, where the sum of the weights in the cut edges is minimal. Since the cut has a minimal weight, the flow between the two partitions should be small, and thus the subgraph in each partition can be ordered individually. When the size of the partition becomes smaller than a given threshold, the optimal layout for the nodes within each partition is solved via exhaustive search. Our implementation sets the threshold as 4, which yields 4! permutations to test.

The algorithm above is primarily designed for undirected graphs, while the ADG used in our problem is directed. To accommodate this, when we partition a directed graph, we first convert it into an undirected graph and then preform the partitioning. To perform the conversion, the weight for each edge in the undirected graph is the sum of the weights in the edges of the original graph that connect the same pair of nodes. Consequently, if there hardly exists flow between a pair of two nodes, their edge weight in the converted graphs will be small too.

To further enforce that the final block layout can represent the flow direction in the directed graph other than only being optimal within the local 4 blocks, the bottom-up concatenation of groups of blocks will consider the layout cost again. When concatenating two groups, which can be laid out in two possible orders, the order with smaller cost according to Equation 5 is chosen. This bottom-up reordering is performed till the root level.

4.4 Visual Analysis of Layouts

Before we discuss the real runtime performance of our layouts, here we use a 2D dataset to visually demonstrate the benefit of the resulted layouts. The dataset is plotted in Figure 3 (a), which is a 2D slice of the dataset *Isabel* where the flows mostly lay in the X-Y direction. Figures 3(b)-(e) show the layouts optimized for graphs generated by different numbers of hops. The line segments connect the nodes in the layout order. Line segments that connect non-adjacent blocks are drawn as thinner lines to avoid visual clutter of the visualization. The color of each end point of the line segments reflects the layout order of the related block. The color map is shown in Figure 3 (f).

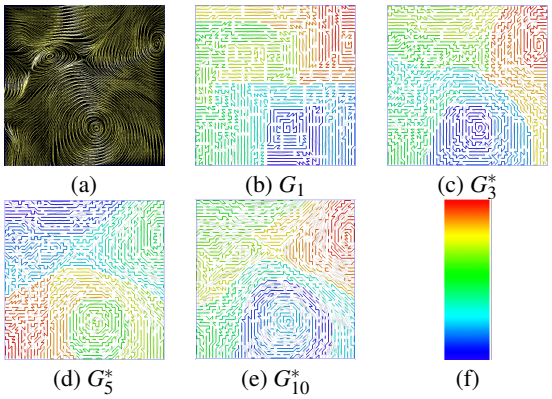

Figure 3: The 2D layout of a slice from *Isabel*. (a): Streamlines showing the flow field, which exists one source on the right side and two sinks at the left corners. (b)-(e): The layouts generated by using the multi-hop graphs of G_1, G_3^*, G_5^*, and G_{10}^*. The color of each end point of the line segments reflects the layout order of the related block. (f): Colormap of the layout order (from the bottom color to the top color)

From Figure 3(b) where the layout was generated from G_1, we can clearly see that the graph is partitioned along the X or Y axis. This is because during graph construction, most particles will move to their neighbors in either X or Y direction. Therefore the generated graph edges are mostly axis aligned. For such a graph, the partition boundaries are usually axis aligned because cutting along the diagonal direction will take two edges and therefore has higher cost. We can see that the layout does not follow the flow directions as shown in Figure 3(a).

In Figures 3 (c)-(d), we can see that the created layout starts to follow the flow directions better since the partition boundaries match the streamlines in Figures 3 (a) than Figures 3 (b). By closely inspecting the layout orders in Figure 3 (e), nevertheless, we find that the layout for G_{10}^* is more fragmented and a larger number of distant jumps occur. As this fragmented layout can be related to the complexity of the graph, it might cause higher miss ratio. We will further quantitatively evaluate the layouts in Section 6.2.

5 Out-of-core System

In this section we present our out-of-core streamline computation system with a prefetching mechanism, using the reordered flow field file.

Our system uses two threads, an I/O thread and a streamline computation thread. Associated with each thread there is a queue managing the computation or I/O requests, as shown in figure 4. Both queues pop out tasks in a FIFO (first-in-first-out) order. During particle tracing, the computation thread checks whether the data block needed for the current particle location is in the memory pool. If not, a request for loading the data is pushed into the I/O queue, and the particle is temporally suspended. The requests in the I/O queue are handled by the I/O thread where the requested blocks are loaded into main memory. When the I/O is completed, the suspended particles are pushed back to the computation queue and is ready to proceed. Note that instead of issuing an I/O request for every suspended particle, we merge requests for the same data block into a single request. This is done by first checking if there is already a request for the same data block. This way, we can avoid loading the same data blocks multiple times.

When the particles of the current blocks are all suspended, the computation thread checks the next block in the computation queue. Essentially, our current scheduling scheme advects the particles in a round robin manner. It should be noted that as more particles are advected in core, more entries in the memory pool will be needed.

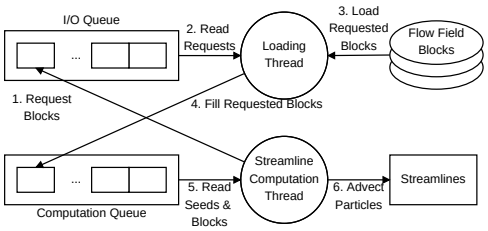

Figure 4: Out-of-core visualization system.

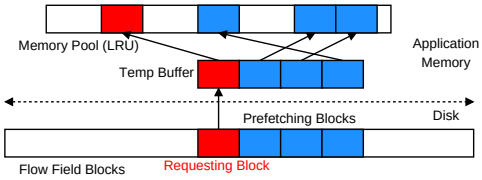

Figure 5: Block prefetching. The blocks are firstly loaded into a temporary buffer and then separately copied into the memory pool.

Because the memory pool has a limited size, the number of particles in core can impact the utilization of the memory pool and thus the runtime performance. As we can divide the particles into groups and advect the particles in batches to maintain the efficiency of the memory pool, Section 6 shows how the number of in-core seeds influence the performance.

The I/O thread maintains a memory pool with space that is large enough to store a certain number of data blocks. Given an I/O request, as shown in Figure 5, the I/O thread prefetches a sequence of contiguous blocks from disk following the required data block in a single I/O request. We use standard C functions such as *fread* for file I/O and a temporary buffer to hold the acquired blocks. Each of the blocks is copied to the memory pool using *memcpy* when space in the memory pool is available. If the memory pool is full, the block that is not recently used by the computation thread is released, based on the LRU (Least Recently Used) order. Because LRU does not guarantee contiguous space to be released, after some time the memory pool can become fragmented. However, in order to prefetch multiple blocks via a single I/O request, a contiguous chunk of memory is needed, and thus a temporary buffer is used to store the loaded data blocks temporarily, as illustrated in Figure 5. Because the size of the temporary buffer is the prefetching size in blocks multiplied by the size of a single block, the storage overhead is smaller than the memory pool.

6 Results

We evaluated our algorithm based on theoretical analysis of the graph model and the actual running time from streamline computation. We used three datasets in our test: *Nek*, *Isabel* and *Plume*. *Nek* is generated from the fluid dynamics solver NEK5000 from Argonne National Laboratory, whose dimension is 512^3. *Isabel* is taken from the benchmark data for IEEE Vis Design Contest 2004, which simulates hurricane Isabel from September 2003 over the west Atlantic region. We used the first time step of the data set in our test, and upsampled it two times per dimension into 1000 x 1000 x 200. *Plume* is a simulation of the thermal down-flow plumes on the surface layer of the sun. The dimension is 512 x 512 x 2048.

6.1 Performance for preprocessing

In our experiments, the ADG graph was constructed as follows. Each graph node represents a block of size 16^3, and the graph edge weight is computed by placing one particle per $2 \times 2 \times 2$ voxels, i.e., each block receives $8 \times 8 \times 8$ seeds. For consistency, our system

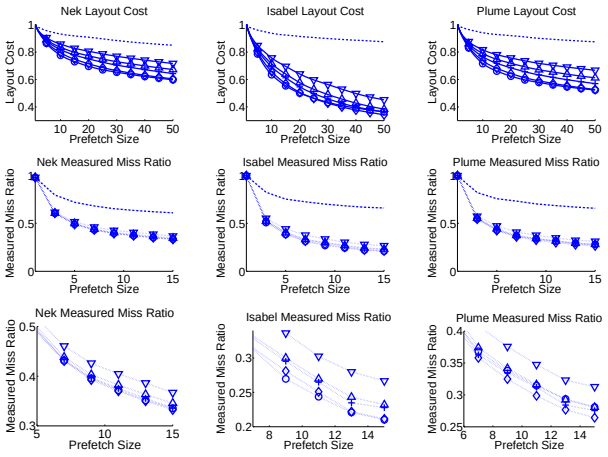

Figure 6: Layout validation. (a): Layout costs of $EMR(L, G_{10}^*)$ in Equation 4 with different prefetch sizes. (b): Averaged miss ratio of single particle tracing. (c): A close view of (b). The curves are for H-curve (...), the optimized layout for G_1^*, ($\bigtriangledown$), G_2^* ($\bigtriangleup$), G_3^* ($+$), G_5^* ($\diamondsuit$), and G_{10}^* ($\bigcirc$).

uses the Runge-Kutta 4^{th}-order integration scheme for both graph construction and streamline computation. The distance that each seed will travel depends on the hops allowed for the graph. Additionally, a maximal advection step is chosen to avoid infinite looping of streamlines in a block. We use several computation threads and each thread advects seeds within a block. After all the seeds hit the block boundary, seeds moving to the same neighboring block are grouped and pushed to a computation queue. Meanwhile the ADG is updated, as the number of seeds in each group represents the edge weight of the related blocks. The threads keep processing jobs in the queue until the queue is empty. The experiment was done by using eight computation threads on a Linux workstation with an Intel Xeon CPU, 24 GB of RAM, and a RAID-5 disk array. Note that the ADG construction was performed in-core since we used the machine with larger RAM than each of our datasets.

Table 1 shows the performance of the preprocessing stage, including graph construction, layout generation and file reordering. As we can see, the graph construction is the most time consuming part in the preprocessing stage. To further understand the bottleneck in this part, we measured timing for each component for each thread, including streamline integration and updating ADG. Our experiment for each dataset shows that between 94% and 98% of the ADG generation time is spent on streamline integration, since it processes a large amount of particles. As the streamline integration can be done locally without inter-process communication, we expect to gain effective speedup when using CPU with more cores or GPGPUs. We leave the improvement of this part as the future work.

6.2 Evaluation of Miss Ratios and Layout Cost

The performance of the streamline computation relies on both the layout of the input flow field and the out-of-core system. In this section we evaluate the quality of the file layouts first, based on the prefetch miss ratios, while the next section will discuss the runtime performance of the file layout with the out-of-core system. To measure the miss ratios, we randomly placed a seed in the flow field and count the number of data misses incurred when advecting this seed. By data misses we mean data blocks that are needed by the particle to advect but have not been prefetched into main memory. The miss ratio is calculated as the number of misses over the total number of blocks that are used for advection. The tests ran for over 300 times

Table 1: Statistics of ADGs and performance in preprocessing stage (The four numbers in columns 5 - 7 are for $G_1/G_3^*/G_5^*/G_{10}^*$).

Dataset	File Size	Dimension	# Blocks	# Edges (in millions)	ADG Generation	Layout Generation	Reordering
Nek	2.28 GB	512^3	33K	0.1/0.7/1.2/2.8	17/58/99/201 min	8/13/22/44 sec	58 sec
Isabel	3.42 GB	$1000^2 \times 200$	52K	0.1/0.4/0.7/1.7	20/75/132/264 min	12/15/18/29 sec	148 sec
Plume	8.88 GB	$512^2 \times 2048$	131K	0.3/1.9/3.1/6.4	49/190/338/712 min	37/125/140/221 sec	308 sec

with different seed locations. The average miss ratios are plotted in the middle row of Figure 6. It can be seen that our algorithm creates layouts with smaller miss ratios than Hilbert curve. Besides, from the zoomed in views in the bottom row of Figure 6, we can see that the layouts based on multiple-hop graphs have smaller miss ratios than those based on single-hop graphs.

We did another test to study the relation between our cost function used for the layout algorithm and the real miss ratio. The top row of Figure 6 shows the costs of the computed layouts, where the cost is computed according to Equation 5 by using ADG G_{10}^* with different prefetch sizes. It can be seen that the evaluated costs of layouts optimized for ADG G_5^* is very close to the layout optimized for G_{10}^*. One reason is that after 10 hops, the seeds are widely spread over the entire domain. Consequently, new edges created by allowing particles to go further are of small weights and therefore play a limited role in affecting the file layout.

6.3 Run time performance

In this section we show the runtime performance of streamline computation by using our prefetching scheme. To study the performance gain, we tested different parameters for each dataset including prefetch sizes, seeding scenarios, and the selection of hops used to generate the layouts.

The tests were run on a linux desktop with a single 7200 rpm, 400 GB disk (model: HITACHI Deskstar 7K400) and 1GB of RAM. The CPU used in the test was AMD Athlon 64 - 3500. In all tests, we set the memory pool with a limit of 6250 blocks, taking memory usage within 500 MB. Similar to the previous section, the layouts for comparison were H-curve and our layouts optimized for G_1, G_3^* and G_5^*. Layouts for G_{10}^* were not considered for the run-time test since as shown in figure 6, the miss ratios using the G_{10}^* layout were not improved, and the ADG construction time was much longer, as shown in table 1. The streamlines were computed using the Runge-Kutta 4th order algorithm with fixed step size of 1 voxel. The generated streamlines were stored in memory, so there was no disk write. We measured the timings for computation and data transfer, and the number of requested and loaded blocks. The data transfer time was accumulated from the return time of *fseek*s and *fread*s for all blocks.

Based on different scenarios where scientists would like to understand the flow field, we tested the computation performance in two different seeding strategies. The first one is *uneven seeding* that seeds all the particles in a local region, which mimics a cubic probe used to visualize particles originated from a specific region. The second one is *uniform seeding* that evenly seeds the initial particles in the entire domain, which can help the viewer to obtain a global overview of the flows.

Another parameter for seeding is seed density. In our tests we chose different density with $64(4^3)$, $512(8^3)$, and $4096(16^3)$ seeds. We tested the layouts optimized for G_1, G_3^* and G_5^*, and compared them with the Hilbert-curve layout.

6.3.1 Uneven Seeding Results

In this test, we put seeds in several randomly selected regions. We fixed the region size to 32 voxels per dimension and tested the runtime performance with different number of seeds. Every seed ran at most 4000 steps or until they went out of the domain bound or trapped around critical points. Figure 7 shows the average number

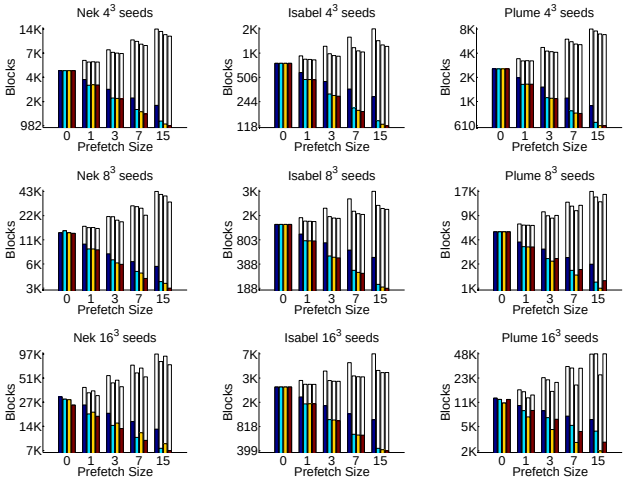

Figure 7: The number of missed blocks (color bars) and the total number of loaded blocks (white bars) for uneven seeding. Note that the box counts are plotted in logarithm scale. The bars in each group from left to right represent the result from Hilbert-curve layout and our layouts optimized for G_1, G_3^* and G_5^*.

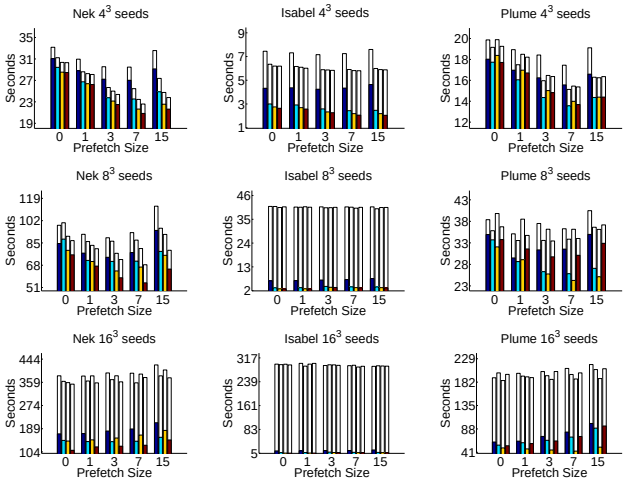

Figure 8: The block loading time (color bars) and wall time (white bars) of uneven seeding. The bars in each group from left to right represent the result for Hilbert-curve layout and our layouts optimized for G_1, G_3^* and G_5^*.

of cache-missed blocks (color bars) and preloaded blocks (white bars) during streamline computation for each dataset.

From the number of cache-missed blocks in the plots, we can see that when using larger prefetch sizes, the number of cache-missed blocks was reduced. Within the same prefetch size, our layouts had a smaller number of cache-missed blocks than that of H-curves. We can also see fewer cache-missed blocks for layouts optimized

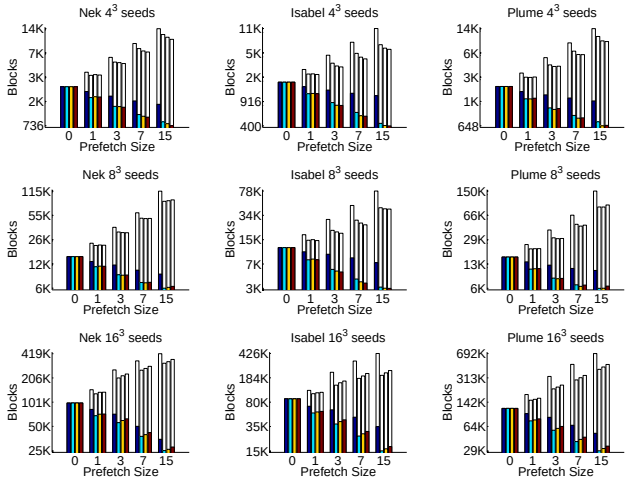

Figure 9: The number of missed blocks (color bars) and the total number of loaded blocks (white bars) for uniform seeding. Note that the box counts are plotted in logarithm scale. The bars in each group from left to right represent the result from Hilbert-curve layout and our layouts optimized for G_1, G_3^* and G_5^*.

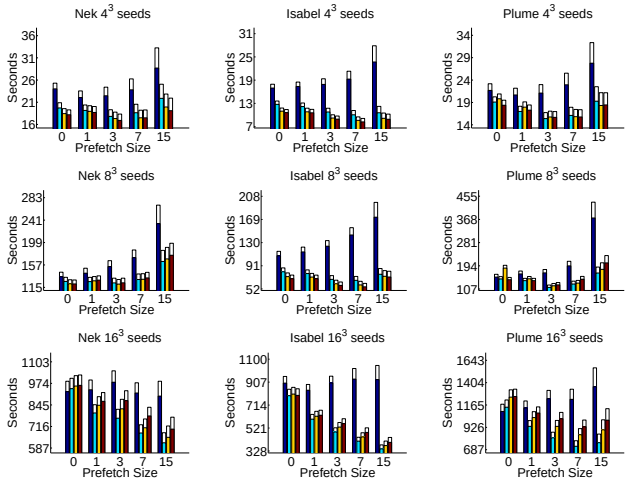

Figure 10: The block loading time (color bars) and wall time (white bars) of uniform seeding. The bars in each group from left to right represent the result for Hilbert-curve layout and our layouts optimized for G_1, G_3^* and G_5^*.

for ADGs with higher hops in 4^3 seed size and 8^3 seed size except for the *Plume* dataset. In 16^3 seed test, nevertheless, the layouts for G_5^* might not always have the smallest miss ratios. Our hypothesis is that when larger seed size is used, more blocks are needed simulatnesouly, and thus the size of the memory pool also influences the miss ratios. We will discuss this issue more in the next section.

Figure 8 shows the I/O performance of uneven seeding. The color bars represent the accumulated data loading time. The top edge of the white bars marks the overall run time calculated by differencing the start and the end of the wall clock time. As can be seen, in most cases when the prefetch sizes are larger than one, our layouts require less data loading time than the Hilbert-curve layout.

The overall runtime also correlates to the data loading time, except for the dataset *Isabel*. From the block counts in the middle row of Figure 7, we can see that given 8^3 seeds, *Isabel* requested fewer blocks than other datasets. In other words, the I/O overhead is smaller, and thus, the total runtime is dominated by the computation time, implying that, the performance cannot benefit from the reduced I/O time. Similarly, the result of 16^3 seeds show that computation time dominates the overall run time, as more computation is needed for the same amount of data to load.

Otherwise, from the results of 4^3 seeds and 8^3 seeds, we can empirically determine that the best prefetch size is 7 blocks, and the optimal ADG to use for the layout is G_5^*. The overall performance gains are listed in Table 2 (a).

6.3.2 Uniform seeding

Here we discuss the performance for uniform seeding. Figure 9 shows the counts of cache-missed blocks and loaded blocks for uniform seeding in each dataset. From the plots of block counts, we can see that when a larger prefetch size is used, the number of cache-missed blocks reduces. Our layouts always have smaller miss counts compared to that of H-curves for the same prefetch size larger than zero. Besides, the layouts optimized for either G_3^* or G_5^* give fewer misses than the layouts for G_1 in the 4^3 and 8^3 seed tests. On the contrary, the layouts for G_1 give fewer misses in 16^3 seed tests.

The results for the 16^3 seed test is related to the memory pool size and the loading method. In our tests, the memory pool could only store 6250 blocks. As the seeds are uniformly placed in the flow field, each seed will request a different block. When the prefetch size is larger than 1, to compute 4096 seeds the loader needs to prefetch more than 8192 blocks, which exceeds the capacity of our memory pool. In such a case, the previously prefetched blocks are quickly flushed out by new requests, leading to poor utilization of the prefetched data that may be useful after several rounds.

Figure 10 shows the performance of uniform seeding. Similar to the uneven seeding scheme, we can see that when the prefetch size is larger than one, our layouts require less loading time than H-curve layout. It can also be seen that unlike the uneven seeding performance of *Isabel* in Figure 8, now the wall clock time for all datasets are also smaller using our layouts than the H-curve layout. One reason is that because the seeds are spread over the entire domain, as a result, more blocks need to be loaded in memory, leading to higher I/O overhead and thus the benefits of our system become more apparent.

From these tests, we can empirically decide the optimal parameters for the file layout and the prefetch size. When the prefetch size is fixed, our layouts optimized for G_3^* or G_5^* usually provides the smallest loading and computation time compared to other layouts. Regarding the prefetch size, prefetching 7 blocks usually provides a better performance. We compute the performance gain by comparing layouts for G_3^* with prefetching 7 blocks with H-curve layouts without prefetching. The results are listed in Table 2 (b).

To summarize, from the above tests, we conclude that our loading scheme works most efficiently when prefetch size is 7 blocks. Meanwhile using the file layout optimized for G_3^* or G_5^* takes less I/O time than the layout using G_1 or Hilbert-Curve layout. While the ADG with more hops provides more information about block dependencies, as discussed in section 6.2, constructing an ADG with too many hops will take too much time. Besides, the corresponding layout can have a higher miss rate. For this, our hypothesis is that after several hops, the seeds tend to scatter out over the entire domain and hence produce many edges with small weights. Our current cost function in Equation 5 does not consider the scattering of seeds and thus can overemphasize the edges with small weights, leading to less optimal layout. We leave the study of the appropriate hop selection in our cost model as a future work.

Regarding the benefit of our layout to the overall run time performance, it depends on whether the I/O time or the computation time dominates. In addition, the runtime performance also depends

Table 2: Speedups in wall time by prefetching 7 blocks in our system, compared to H-curve layout without prefetching. (a): Layouts for G_5^* in uneven seeding. (b): Layouts for G_3^* in uniform seeding.

	(a) Uneven Seeding			(b) Uniform Seeding		
Dataset	Nek	Isabel	Plume	Nek	Isabel	Plume
4^3 Seeds	46%	28%	32%	32%	88%	33%
8^3 Seeds	55%	1%	12%	2%	78%	16%
16^3 Seeds	-4%	2%	-1%	30%	98%	29%

on the size of allocated memory pool, which should be managed more carefully when scheduling the particles for advection. The scheduling issue will be further discussed in the next section.

7 Limitations

While our data prefetching scheme and file layout perform better than the Hilbert-curve layout, our system has some limitations.

Our ADG model can be further improved. Because the ADG model approximates the flow field at the block-level, the approximation error can affect the miss rates of the file layout. Thus in the future we will study the parameters related to the accuracy, including the selection of block size and sampling seeds to construct the ADG. First, an appropriate block size should be chosen, as graphs with too large block size cannot capture the fractal flow transport in smaller scale, while using too small block size might introduce extra storage overhead. As our main goal is to minimize the miss rate, which is more related to the dependencies among the blocks, if the blocks contain more complex flow patterns, the streamline computation time in each block can increase, which will become larger than the I/O time. Second, the selection of sampling seeds can be made more effectively. Currently we use dense seeds to sample the edge weights, which lead to a heavy preprocessing cost. Using adpative sampling to reduce the number of sampling seeds while preserving the accuracy of flow graph should be studied too. Because of the preprocessing cost, for datasets of extreme scale where reordering done as a post process is undesired, our algorithm is not suitable unless the preprocessing cost can be further reduced.

Other limitations are mainly related to our out-of-core system, especially the scheduling of particle tracing. Our current implementation uses FIFO to advect the seeds in the computation queue, which can be further improved. As the I/O is the main performance bottleneck, the scheduler should carefully schedule the particles to avoid redundant block loading. Besides, as the size of the memory pool is limited, the scheduler should utilize the memory more efficiently too; otherwise, if too many particles are executed in core, the data in the memory pool will be frequently flushed and thus become under-utilized, reducing the effectiveness of the prefetching scheme, as demonstrated in Section 6.3.2.

8 Conclusions

We have presented a novel data layout algorithm and a prefetching scheme for out-of-core streamline computation. With the knowledge of flow directions, our layout allows more efficient I/O than Hilbert-Curve layout when performing out-of-core streamline computation. The advantage of our approach can be attributed to the knowledge of data access pattern derived from flow directions in the given flow field, which in turn allows us to perform more effective data prefetching.

In our future work, first we plan to investigate how to decide an optimal prefetch size based on the machine in use automatically. Additionally, we will adapt the ADG model for time-dependent flow fields. For time-dependent flow fields, as each time step is stored separately, we should investigate whether the blocks from different time steps should be laid out together, or should mainly rely on the scheduling scheme to reduce the I/O overhead. Also, related visualization techniques for time-dependent flow fields, including pathlines, streaklines/streaksurfaces and FTLE, should be considered. All of them require efficient scheduling schemes for broader applications.

Acknowledgements

The authors would like to thank the anonymous reviewers for their comments. 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] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics Forum*, 21(2):137–149, 2002.

[2] R. Bruckschen, F. Kuester, B. Hamann, and K. Joy. Real-time out-of-core visualization of particle traces. In *PVG '01: Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, pages 45–50, 2001.

[3] G. Chen, K. Mischaikow, R. S. Laramee, P. Pilarczyk, and E. Zhang. Vector field editing and periodic orbit extraction using morse decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):769–785, 2007.

[4] M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Vis '97: Proceedings of the IEEE Visualization 1997*, page 235, 1997.

[5] M. Griebel. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 25(7):827–843, July 1999.

[6] M. Isenburg and P. Lindstrom. Streaming Meshes. In *VIS '05: Proceedings of the IEEE Visualization 2005*, pages 231–238, 2005.

[7] Z. Karni, A. Bogomjakov, and C. Gotsman. Efficient compression and rendering of multi-resolution meshes. In *Vis '02: Proceedings of the IEEE Visualization 2002*, pages 347–354, 2002.

[8] T. Leighton and S. Rao. Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[9] R. Niedermeier and P. Sanders. *On the Manhattan-Distance Between Points on Space-Filling Mesh-Indexings*. Univ., Fak. für Informatik, 1996.

[10] 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*, 2011. To appear.

[11] V. Pascucci and R. Frank. Global Static Indexing for Real-Time Exploration of Very Large Regular Grids. In *SC '01: Proceedings of the ACM/IEEE 2001 Conference on Supercomputing*, pages 45–45, 2001.

[12] C. T. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. *Out-Of-Core Algorithms for Scientific Visualization and Computer Graphics*. 2002. IEEE Visualization '02 Course Notes.

[13] P. Sulatycke and K. Ghose. A fast multithreaded out-of-core visualization technique. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 569–575, 1999.

[14] M. Tchiboukdjian, V. Danjean, and B. Raffin. Binary mesh partitioning for cache-efficient visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):815–28, 2010.

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

[16] L. Xu and H.-W. Shen. Flow Web: a graph based user interface for 3D flow field exploration. In *Proceedings of IS&T/SPIE Visualization and Data 2010*, volume 7530, page 13, 2010.

[17] S.-E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–20, 2006.