Component-based Data Layout for Efficient Slicing of Very Large Multidimensional Volumetric Data

Jusub Kim
Institute for Advanced Computer Studies
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742
E-mail: jusub@umd.edu

Joseph JaJa
Institute for Advanced Computer Studies
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742
E-mail: joseph@umiacs.umd.edu

***Abstract*— In this paper, we introduce a new efficient data layout scheme to efficiently handle out-of-core axis-aligned slicing queries of very large multidimensional volumetric data. Slicing is a very useful dimension reduction tool that removes or reduces occlusion problems in visualizing 3-D/4-D volumetric data sets and that enables fast visual exploration of such data sets. We show that the data layouts based on typical space-filling curves are not optimal for the out-of-core slicing queries and present a novel *component-based data layout* scheme for a specialized problem domain, in which it is only required to provide fast slicing at every k-th value, for any $k > 1$. Our component-based data layout scheme provides much faster processing time for any axis-aligned slicing direction at every k-th value, $k > 1$, requiring less cache memory size and without any replication of data. In addition, the data layout can be generalized to any high dimension.**

I. Introduction

There is a consistent trend in almost all scientific and medical domains toward increasingly generating higher resolution data as computing power steadily increases, and sensor and imaging instruments get more refined. The high resolution data sets are often generated and stored as 3-D or 4-D volumetric data or sometime as even higher dimensional data sets. Scientists and engineers often study physical phenomena by simulating their mathematical models on supercomputers, thereby generating time-varying volume data sets of sizes ranging from hundreds of gigabytes to tens of terabytes. Also, biomedical equipments such as CT, MRI, and 3-D confocal microscopy are now capable of providing very high resolution volumetric data. For example, the visible human project [1] produced volumetric data up to 40 GB representing complete normal adult anatomy.

However, the interactive visual exploration of high resolution data sets currently presents substantial challenges, especially when the data cannot fit in main memory. Efficient data transfer from disk to main memory is critical in achieving an efficient visual exploration of the very large data sets. The performance of the algorithms that have to deal with the data stored on disks is very much determined by the data layout and the corresponding access patterns because of the sequential access property of disks.

In this paper, we introduce a new efficient data layout scheme to efficiently handle out-of-core axis-aligned slicing queries of very large multidimensional rectilinear grids. In n-dimensional volumetric data, we define axis-aligned slicing as the process of obtaining a (n-1)-dimensional slice by taking the sample points on the n-dimensional plane $I_j = \alpha$, where $(I_1, I_2, ..., I_n)$ are the dimensions and α is one of the grid values along the jth dimension. Slicing is a very useful dimension reduction tool because it removes or reduces occlusion problems in visualizing 3-D/4-D volumetric data sets and it enables fast visual exploration of such data sets. While a typical example is the display of 3-D MRI or CT volumetric data using 2-D slices, sequentially rendering each time step of a time-varying volume data set is the case of slicing a 4-D volumetric data set. When data is too large to fit in main memory, we should only load the data relevant to the particular slicing query. In this paper, we show that the data layouts based on the typical space-filling curves such as Z-order or Peano-Hilbert order [2] are not optimal for the out-of-core slicing queries, and introduce *component-based data layout* scheme in a specialized problem domain, in which it is only required to provide fast slicing at every k-th value, $k > 1$.

Our scheme has four key features. First, it provides much faster processing time for any type of axis-aligned out-of-core slicing queries at every k-th value, $k > 1$, than any of the typical space-filling curves approaches. And, the performance gap widens as we deal with

larger data. Second, it requires less cache memory size, given the same data blocking factor. This is especially desirable when we have to deal with large data in a very limited system environment such as a laptop computer with small amount of main memory. Third, it does not replicate any of data. A typical solution to enable fast slicing along every axis is to replicate data and store each copy using a different lexicographical layout scheme. However, our scheme is based on the divide and conquer approach that avoids any replication of any part of the data. Last, our data layout is mathematically defined for any dimension and any value of $k > 1$, and its performance can be analyzed analytically.

We use this scheme to visually explore the visible human male data set [1]. With the new approach, we can interactively slice the extremely large 3-D volumetric data ($\sim$2 GB) in every direction achieving around 3$\times$ and 10$\times$ performance improvements respectively at full and half resolution while requiring 22% and 88% less cache memory size than a typical Z-order scheme.

In the rest of paper, we first review related work followed by introducing our new data layout scheme. The analysis of our scheme and related experimental results are then presented. We conclude with a brief summary and plans for future work.

II. Related Work

Due to their electromechanical components, disks have two to three orders of magnitude longer access time than random access main memory. A single disk access reads or writes a block of contiguous data at once. The performance of an out-of-core algorithm [3] is often dominated by the number of I/O operations, each involving the reading or writing of disk blocks. Hence designing an efficient out-of-core visualization algorithm requires a careful attention to data layout and the organization of disk accesses in such a way that necessary data blocks are moved in large contiguous chunks into main memory.

A number of out-of-core techniques to handle a variety of scientific visualization [4] problems have appeared in the literature as larger and larger data sets are being generated. Cox and Ellsworth [5] show that application-controlled paging and data loading in a unit of subcube with the ability of controlling the page size can lead to better performance in out-of-core visualization. Out-of-core isosurface extraction algorithms are reported in [6], [7], [8]. Out-of-core volume rendering algorithms are reported in [9], [10], [11]. Silva et al. [12] provide a good survey on out-of-core algorithms for scientific visualization and computer graphics.

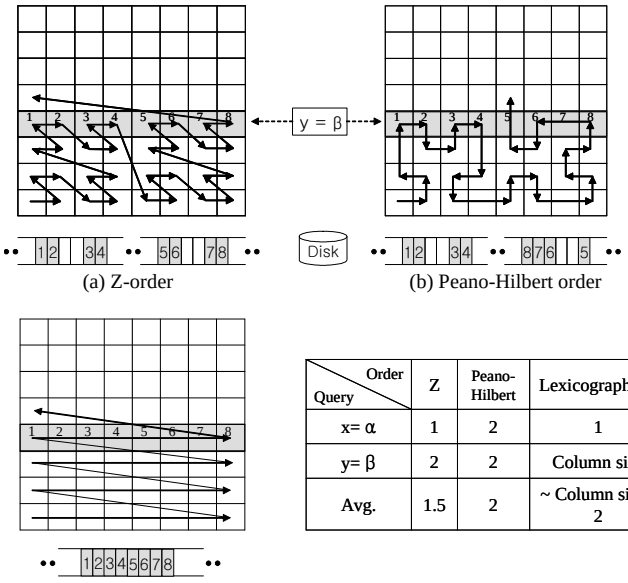
For efficient out-of-core data accesses, it is important to lay out data in a way that various access methods retrieve the data sequentially. Space filling curves [2] have been used for mapping n-dimensional data to 1-dimension while trying to preserve the spatial locality of the original n-dimensional data. The most popular ones are the Z-order [13] and the Peano-Hilbert order [14]. While the Peano-Hilbert order has a slightly higher degree of locality than the Z-order, the Z-order has been more frequently used because of the simplicity of the conversion process between the key and its corresponding element in the multidimensional space. Recently, Lawder [15] examines different kinds of space filling curves to develop indexing schemes for fast retrieval of data in multi-dimensional databases. Pascucci and Frank [16] present a variation of the Z-order for progressive traversal and visualization of large regular grids. They combine interleaved storage of the levels in the data hierarchy while maintaining geometric proximity within each resolution level.

III. Efficient Data Layout for Slicing Query

The widely used space filling curves such as Z-order and Peano-Hilbert order [2] store neighboring multidimensional data as sequentially as possible in storage, and hence they have been widely used because they provide a good cache efficiency for accessing n-dimensional data. While they are effective in the situation where data access occurs across all n dimensions, more efficient data layouts are needed for slicing queries because data access occurs across only (n-1) dimensions for each such query.

For illustration purposes, Figure 1 shows how three different layouts affect the disk I/O efficiency for a slicing query in the 2-D case. Disk I/O efficiency can be expressed by how many contiguous disk pages are accessed for a given query. As shown in the figure, the particular lexicographical-order sequentially stores the 1-dimensional slice corresponding to $y = \beta$ and thus achieves the highest contiguity in disk access for this particular slicing query while the other space filling curves achieve very little contiguity.

However, employing the straightforward lexicographical-order results in worse performance for the least priority slicing query (x=α in the example) although it achieves better performance for the other (n-1) types of slicing queries for the n-Dimensional data. The most naive approach of eliminating the single worst case would be to create an additional copy of the



Query \ Order	Z	Peano-Hilbert	Lexicographical
x= α	1	2	1
y= β	2	2	Column size
Avg.	1.5	2	~ Column size / 2

(d) The number of contiguous disk block accesses

Fig. 1. Data access patterns for a slicing query in a 2-D case for three different data layouts. Grey blocks correspond to the disk blocks satisfying the slicing query y=β.

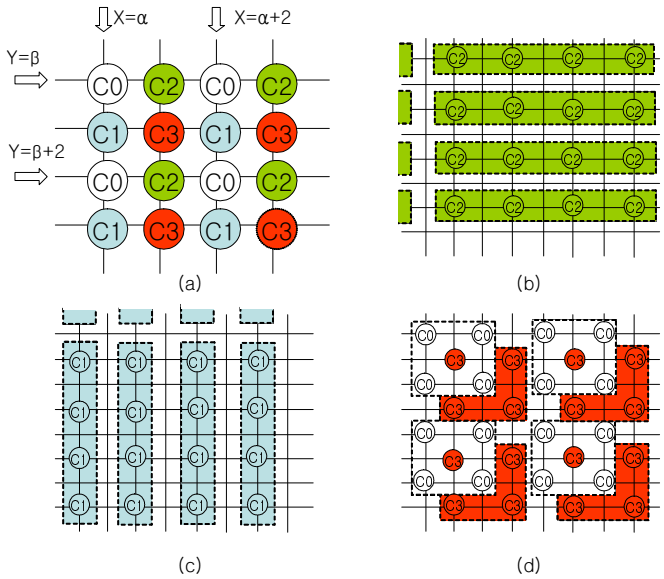


Fig. 2. A 2-D example of the component-based layout for fast slicing at every other value. Components C1 and C2 are grouped into 1-D metacells and stored in the required lexicographical orders while C0 and C3 are grouped into 2-D metacells and stored according to the Z-order. Note that a dotted box indicates each metacell.

data and store that copy in the lexicographical-order in favor of the least priority slicing. However, it is not practical to duplicate the already very large data set.

In the following, we introduce our *component-based layout* scheme to address this problem in a specialized problem domain, in which it is only required to provide fast slicing at every k-th value, $k > 1$. The idea is to divide a n-dimensional data set into non-overlapped components and to systematically provide a different layout that is especially tailored to each component in such a way that isolated disk accesses are eliminated.

A. Case I: k=2

We first explain our component-based layout scheme for k=2, meaning that it is required to provide fast out-of-core slicing only at every other value. Figure 2 shows the layout scheme on 2-Dimensional data for illustration purposes. We divide the 2-D grid into four non-overlapping components, and store each component in the following way. First, we group the elements in component C1 into 1-D metacells and store the corresponding metacells in a lexicographical order that favors the X-slicing since C1 is required only by the X-slicing. Note that a metacell is a block of grid points which is of disk page size. Similarly, we group the elements in component C2 into 1-D metacells and store them in a lexicographical order that favors the Y-slicing. Second, we group the elements in components C0 and C3 into 2-D metacells and store the metacells by Z-order since they are either required by both slicing types or not required by any type of slicing. Now given a slicing query at every other value as shown in the figure, half of the data that we access are always stored in the lexicographical order in favor of that particular slicing, providing maximum contiguous data access, while half of them are stored in Z-order.

Now, we generalize the idea to n-dimensional data. Given a n-dimensional regular grid, let $(i_1, i_2, ..., i_n)$ denote the index of a grid point. Then we define Component-Code (C-CODE) of the index, C-CODE($i_1, i_2, ..., i_n$), as a concatenation of ($i_j \bmod 2$), j=1,2,...,n. Then, we define each component C_i of the n-dimensional regular grid as follows.

$$C_i \equiv \{(i_1, i_2, ..., i_n) | \quad \text{C-CODE}(i_1, i_2, ..., i_n)\text{=i}\}$$

$$Grid_n \equiv \{C_i | \quad i = 0, 1, ..., 2^n - 1\}$$

For example, the C-CODE of the 3-D grid point at (3,2,5) is 5 (=101_2) and thus belongs to component C_5.

Now we define a slicing query as the query to generate the sample points residing on the hyperplane $I_j = \alpha$, where $\alpha \bmod 2 = 0$ (because $k = 2$). Then, a set of necessary components, A_j, for answering the slicing query $I_j = \alpha$ is

$$A_j = \{C_i | \quad \text{the j-th most significant bit of i=0}\}$$

because only the components of which $i_j \bmod 2 = 0$ can be sliced by the plane and there are total 2^{n-1} such components. For example, given a component C_i of which C-CODE is 010_2 in a 3-D grid, we know that it is required for both X- and Z-slicing.

The 2^n components comprising a n-dimensional grid consists of 4 types of components. Let p denote the number of slicing types that a component may be subject to (Note that it is the same as the number of '0's in the C-CODE of the component). Then each component belongs to one of the following types.

1) Type I (p=0): A component that is not required by any type of slicing. There is only one such component such that all the bit values of its C-CODE are equal to '1'.
2) Type II (p=1): A component that is exclusively required by a particular slicing. There are n such components, each of which has a C-CODE having only one bit equal to '0'.
3) Type III ($2 \leq p \leq n-1$): A component that is commonly required by p different types of slicing. Given p, there are ${}_nC_p$ such components.
4) Type IV (p=n): A component that is required by all slicing types. There is only one such component such that all the bit values of its C-CODE are equal to '0'.

For out-of-core access, we store each type of components in the following way.

- Type I and IV (p=0 or n): The elements of each component are grouped into n-dimensional metacells, which are then stored according to Z-order.
- Type II (p=1): The elements of each component are grouped into (n-1)-dimensional metacells, which are then stored in a lexicographical order in a way that the exclusive slicing type gets the highest priority.
- Type III ($2 \leq p \leq n-1$): The elements of each component are grouped into n-dimensional metacells, which are then stored in a lexicographical order in a way that all the p types of slicing get higher priority than the remaining (n-p) types.

Note that for type II, we group elements into (n-1)-dimensional metacells because it is exclusively sliced by a particular slicing type. For type III, we always avoid the worst case, in which any of the p types of slicing gets the least priority in the lexicographical order, since there is always at least one slicing axis that does not require the component and the least priority can be assigned to that dimension.

B. Case II: $k > 2$

Now, we consider the general case where it is required to provide fast slicing at every k-th value, $k > 2$. Figure 3 shows a 2-D example in the case of k=3. Note that the only change is the element size of each component, where the element of each component is the maximal group of contiguous grid points which belong to the same component.

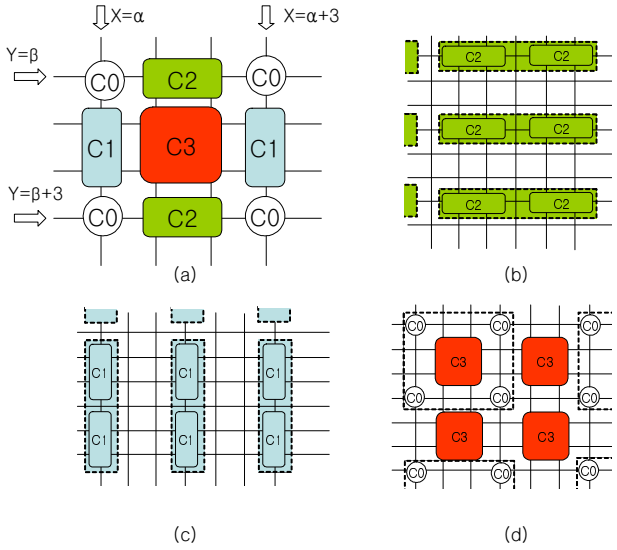


Fig. 3. A 2-D example of the component-based layout for fast slicing at every third value. Note that the only change is the element size of each component.

We redefine a slicing query as retrieving the sample points on the plane $I_j = \alpha$, where $\alpha \bmod k = 0$. Then, we only need to generalize the previous Component-Code (C-CODE) definition as follows.

C-CODE($i_1, i_2, ..., i_n$) $\equiv$ a concatenation of b_j, j=1,2,...,n.

$$b_j = \begin{cases} 0, & \text{if } (i_j \bmod k)\text{=0.} \\ 1, & \text{if } (i_j \bmod k) \neq 0. \end{cases}$$

Now, the element size of each component increases by a factor of (k-1) per a bit value '1' of the C-CODE because (k-1)-times more grid points get included due to $(i_j \bmod k) \neq 0$. Hence, the size increases by a factor of $(k-1)^{n-p}$, where p is the number of '0's and thus n-p is the number of '1's in the C-CODE.

Since the other descriptions in the case of k=2 for n-Dimensional data only depends on the C-CODE of a component, they apply to the general case of $k > 2$ in the same way.

C. Analysis

To analyze and compare the performance of our scheme to other schemes, we define *Contiguity* as the ratio of the average number of disk blocks that can be accessed sequentially to the total number of disk blocks needed for a particular slicing, and *Effectiveness* as the ratio of the amount of data needed to the amount of data transferred. Both indices are equally important in terms of disk I/O cost. In fact, disk access time can be approximated by the the time to read the necessary data at maximum transfer rate $\times \frac{1}{\textit{Effectiveness}}$, plus the time for disk head movement $\times \frac{1}{\textit{Contiguity}}$. Hence, using the two indices, we can compare the component-based layout scheme with the typical Z-order scheme in which the data is first decomposed into n-Dimensional metacells of which size is equal to the disk page and then stored by Z-order.

Let CO_z and EF_z denote the contiguity and the effectiveness of the Z-order scheme while CO_c and EF_c correspond to the component-based data layout scheme. Assuming that an n-D metacell is of size $\underbrace{L \times L \times ... \times L}_{n}$ (and hence the size of a disk block is L^n) and the n-D volume consists of $\underbrace{M \times M \times ... \times M}_{n}$ metacells ($M \gg 2^{n-1}$), a slice of the metacell is of size L^{n-1} and thus the effectiveness of the n-D metacell is always $\frac{1}{L}$ (=$\frac{L^{n-1}}{L^n}$).

The number of sequentially accessed blocks in Z-order is 1,2,4,...,or 2^{n-1} according to the slicing axis. Table I shows the contiguity and the effectiveness of the Z-order combined with the n-D metacell scheme.

	Contiguity (CO_z)	Effectiveness (EF_z)
Z-order	$\bigcirc(\frac{2^{n-1}}{M^{n-1}})$	$\frac{1}{L}$

TABLE I

CONTIGUITY AND EFFECTIVENESS OF Z-ORDER + N-D METACELL SCHEME.

On the other hand, in a lexicographical order in favor of a certain slicing priority, the contiguity becomes 1, $\frac{1}{M}$, $\frac{1}{M^2}$,... in a decreasing order of the priority of the slicing. Table II shows the contiguity and the effectiveness in each type of a component in the component-based layout, assuming that each component is of full volume size. Note that type I components are never required and that type II components have no discontiguous disk accesses and do not load any redundant data.

	Contiguity (CO_c)	Effectiveness (EF_c)
Type II	1	1
Type III	$\Omega(\frac{1}{M^{n-2}})$	EF_z
Type IV	CO_z	EF_z

TABLE II

CONTIGUITY AND EFFECTIVENESS IN EACH TYPE OF A COMPONENT.

Since CO_z is upper bounded by $(\frac{2}{M})^{n-1}$ and CO_c of type III is lower bounded by $\frac{1}{M^{n-2}}$, the value of CO_c for type III components is at least $\frac{M}{2^{n-1}}$ times as high as the CO_z. Thus, type III components always have better contiguity than type IV components of Z-order and the benefit increases as the volume size gets larger.For example, in the case of n=3, i.e., a 3-dimensional volume, type III components have at least $\frac{M}{4}$ times as high contiguity as type IV components of Z-order, i.e., $\frac{4}{M}$ less discontiguous disk head movements than Z-order.

Now given a slicing query, there are total 2^{n-1} components needed to answer the query and among them there is only one component of type II or IV and the other 2^{n-1}-2 components are of type III.

1) Case I ($k = 2$): Since the number of elements of all the 2^{n-1} components comprising a slice is the same, we have

$$\frac{1}{CO_c} = \frac{1}{2^{n-1}} \cdot \frac{1}{CO_z} + (1 - \frac{1}{2^{n-1}}) \cdot \frac{1}{\overline{CO_c}}$$

$$\frac{1}{EF_c} = (1 - \frac{1}{2^{n-1}}) \cdot \frac{1}{EF_z} + \frac{1}{2^{n-1}} \cdot 1$$

(Note that we use harmonic mean for more correct averaging of the two indices. $\overline{CO_c}$ is an average of CO_c for type II and III.)

There is always contiguity improvement over the Z-order scheme, which is upper bounded by 2^{n-1} times as high contiguity, and as n increases, the contiguity improvement gets larger as long as $M \gg 2^{n-1}$. In addition, there is always effectiveness improvement upper bounded by a factor of $\frac{2^{n-1}}{2^{n-1}-1}$. Note that the higher effectiveness also means less cache memory size required for the same slicing query.

2) Case II ($k > 2$): Since the element size of each component of which the C-CODE bit values have p '0's increases by a factor of $(k-1)^{n-p}$, 2^{n-1} is replaced by $R_n(k)$ (=$\sum_{p=1}^{n} {}_{n-1}C_{p-1} \cdot (k-1)^{n-p}$), which is lower

bounded by 2^{n-1} for $k > 2$, then,

$$\frac{1}{CO_c} = \frac{1}{R_n(k)} \cdot \frac{1}{CO_z} + (1 - \frac{1}{R_n(k)}) \cdot \frac{1}{\overline{CO_c}}$$

$$\frac{1}{EF_c} = (1 - \frac{(k-1)^{n-1}}{R_n(k)}) \cdot \frac{1}{EF_z} + \frac{(k-1)^{n-1}}{R_n(k)} \cdot 1$$

Note that the portion that the type IV component of Z-order contributes to the slice decreases while the contribution of type II increases more than any other types since the element size of type II (p=1) increases by the largest factor $(k-1)^{n-1}$ while that of type IV (p=n) does not increase. As a result, we achieve better contiguity and effectiveness as k increases.

IV. Experimental Results

We evaluated the performance of our scheme for k=2, in which it is required to fast process slicing queries at every other value. For the evaluation, we used a subset of the visible human male anatomical image data set [1]. The test volumetric data consists of $2048 \times 1216 \times 800$ grid with 1-byte values, resulting in 2 GB.

We ran all the tests on a single Linux machine which has dual 3.0 GHz Xeon processors with ∼50 MB/s maximum disk I/O transfer rate. In all our experiments, we made use of only one of the two processors. Also, we used a simple buffer management system in order to control disk I/O. The blocking factor for the data was selected arbitrarily to be $8 \times 8 \times 8$.

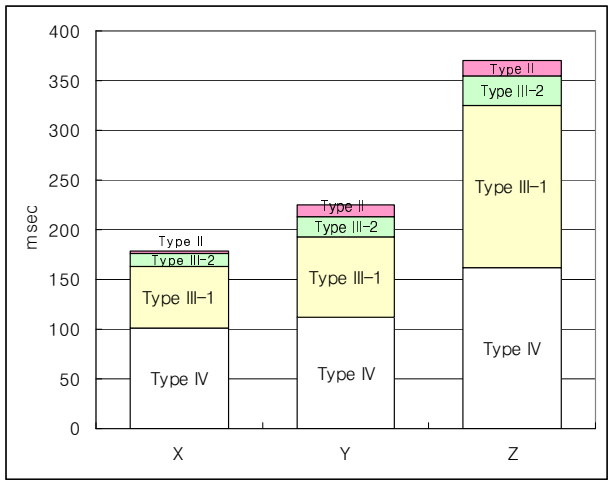


Fig. 4. Contribution of each type of components to total time for X, Y, and Z=α queries.

Figure 4 shows the contribution of each type of components to the total time in performing each type of the slicing queries. Given a slicing query, there are total 4 components required, among which there are only one type II and IV component and two type III components. The type IV component which is stored in Z-order takes the largest 48% of the total time while the type II component which is stored in a lexicographical order in a way that the exclusive slicing type gets the highest priority takes only 4% of the total time. Each of the two type III components takes 40% and 8% of the total time respectively. The Z-slicing takes the longest time because the slice size is the largest.

We compare the performance of the component-based data layout scheme with the Z-order combined with the n-D metacell scheme for three different types of axis-aligned slicing queries.

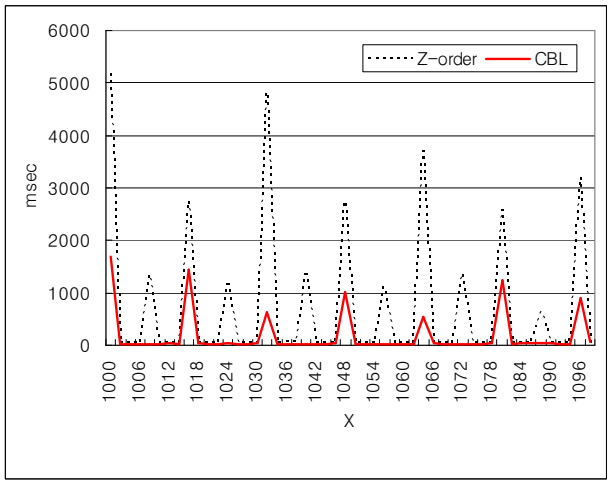


Fig. 5. Performance comparison for loading X=α slices (1216×800).

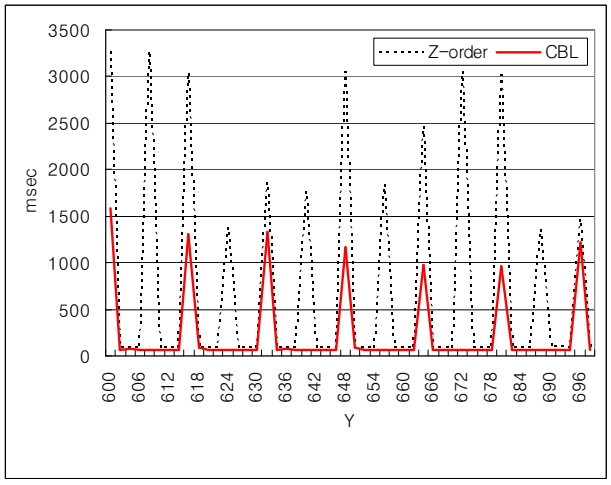


Fig. 6. Performance comparison for loading Y=α slices (2048×800).

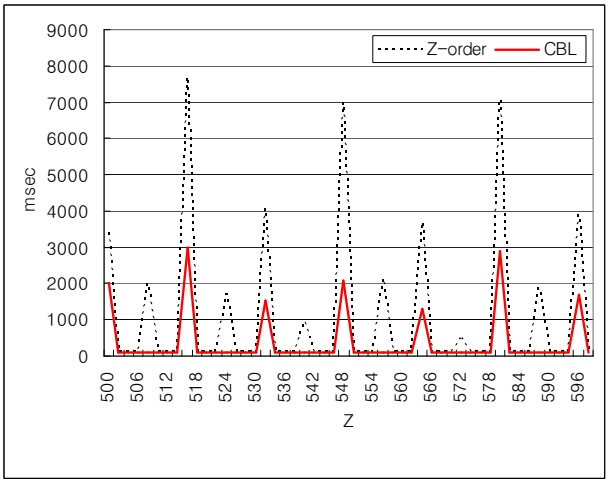


Fig. 7. Performance comparison for loading Z=α slices (2048×1216).

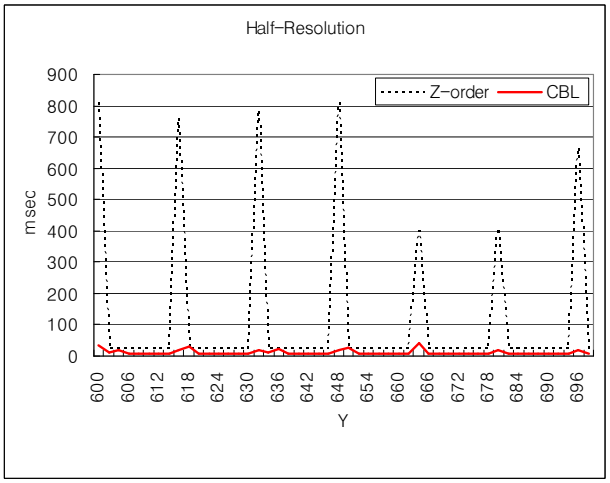


Fig. 9. Performance comparison for loading Y=α slices at half resolution (1024×400).

Figure 5, 6, and 7 compare the total disk I/O time for reading X, Y, or Z=α slices at full resolution. The component-based lay out scheme always achieves better performance, in average by a factor of 3.2. In addition, it requires 16 MB cache memory, which is 22% less than what the Z-order scheme requires. These experimental results are close to the analytical upper bound in our analysis, by which we expect the performance improvement and the cache size reduction to be upper bounded by respectively a factor of 4 and 25%. Note that the same improvements can be expected on any data of the same size given the same blocking factor, since the content of the data is not considered in any of the above process.

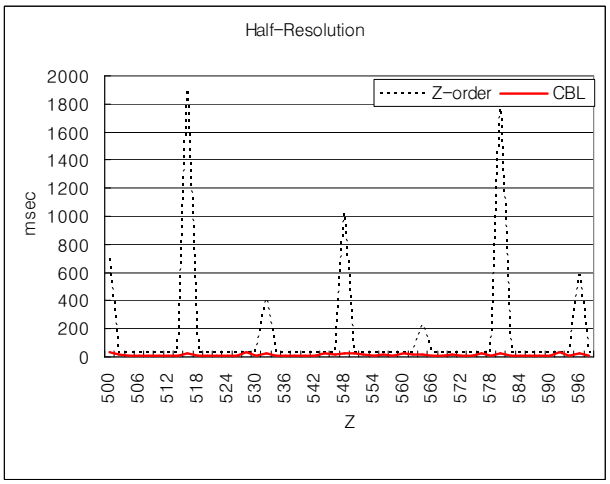


Fig. 10. Performance comparison for loading Z=α slices at half resolution (1024×608).

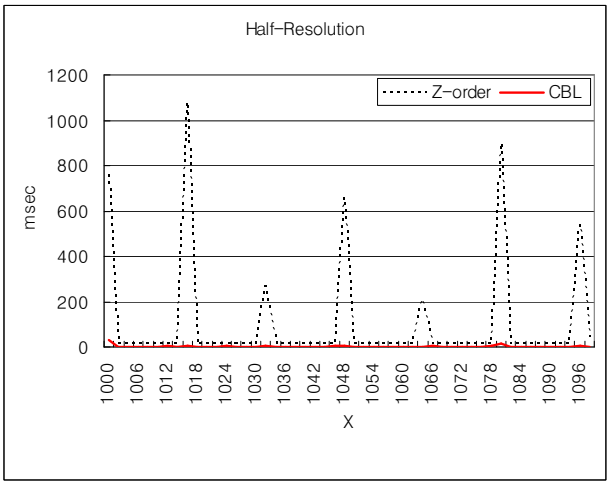


Fig. 8. Performance comparison for loading X=α slices at half resolution (608×400).

For k=2, an additional benefit of the component-based data layout scheme is to be able to perform all types of the half-resolution slicing queries at the maximum disk transfer rate because for every axis-aligned slicing type there is always one half-resolution type II component which is stored in an optimal way for the slicing type. Figure 8, 9, and 10 compare the total disk I/O time for reading X, Y, or Z=α slices at half resolution. The component-based lay out scheme is an order of magnitude faster in average without any performance fluctuation as seen in the figures. In addition, it requires only $\frac{1}{8}$ of the cache memory size for the Z-order scheme, given the particular blocking factor. Note that we compare with the Z-order scheme at half-resolution data (i.e., type IV component).

Figure 11 shows a sample output slice image in each

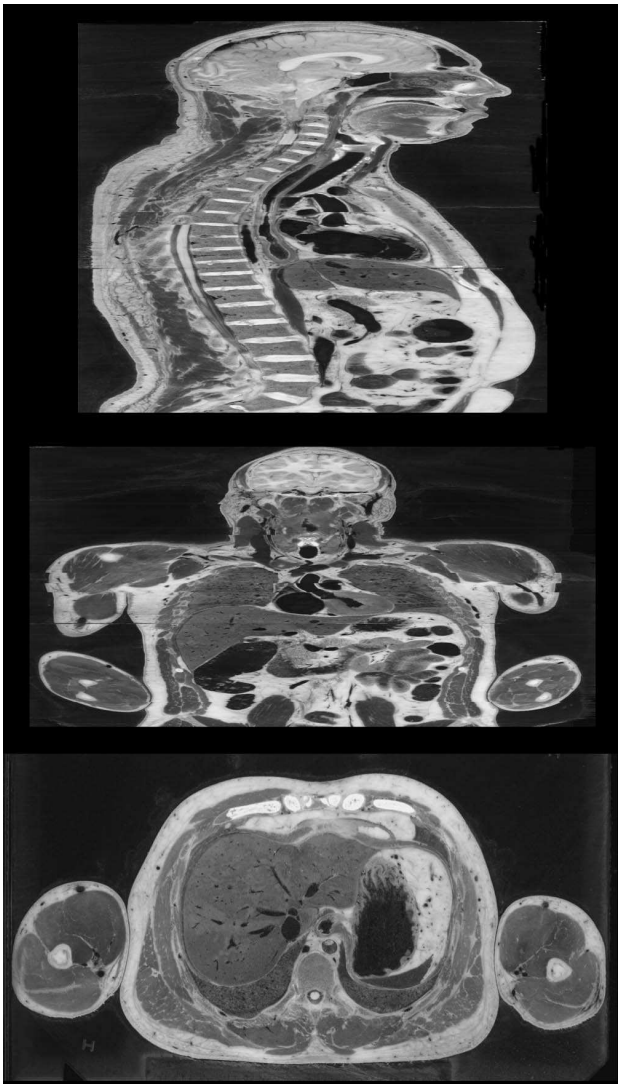


Fig. 11. Sample slice images of the test volumetric data at X, Y, and Z=α. (1216×800, 2048×800, and 2048×1216 from top to bottom.)

type of the slicing queries from our test volumetric data.

V. Discussion

The component-based data layout scheme shows two times larger intervals in performance fluctuation in the timing results. The performance fluctuation is related to the blocking factor. The blocking essentially prefetches the data under the assumption that the slicing queries are given incrementally. While the Z-order + n-D metacells scheme prefetches the data not needed by every k-th value slicing ($k > 1$) as well as the necessary data, the component-based scheme does not prefetch the unnecessary data. Thus it can effectively prefetch larger intervals given the same blocking factor.

Being able to perform the half-resolution queries at maximum disk I/O transfer rate in every slicing type (when k=2) becomes more beneficial when we deal with larger dimension. For a time-series of the test volumetric data, one 3-D slice could easily be of size in hundreds of megabytes to gigabytes. Unless we replicate the already large data, it will be very difficult to achieve the maximum disk I/O transfer rate in all the slicing types by using previous methods. In addition, our scheme requires only the cache memory size equal to the slice size for half-resolution queries.

While the contiguity of type III components is at least $\frac{M}{2^{n-1}}$ times as high as that of type IV components stored in Z-order as shown in the analysis, the performance result in Figure 4 shows that type III components take almost equivalent time (only 8% less) to type IV components of Z-order at the worst case. We believe that this is because the disk head movement time is different between Z-order and lexicographical order. Although Z-order has more discontiguous disk block accesses, the distance between two discontiguous disk blocks is shorter than lexicographical order. In order to investigate this further, we ran all the tests with a $16 \times 16 \times 16$ blocking factor, which is 8 times bigger disk page size. And we observed 60% less time in type III components compared to type IV components at the worst case. Overall, the blocking factor change results in 23% less time in component-based scheme and 15% less time in the Z-order scheme due to two times large cache memory, but with worse peak processing time. Performance improvement was slightly bigger 3.5 at full resolution.

VI. Conclusion

In this paper, we have presented a new data layout scheme to efficiently handle out-of-core axis-aligned slicing queries of very large multidimensional rectilinear grids. We have analytically shown that our scheme provides faster processing time and requires less cache memory than the typical Z-order scheme for any type of axis-aligned out-of-core slicing queries at every k-th value ($k > 1$), without any data replication. Through experimental results, we have also demonstrated that it could achieve $3\times$ and $10\times$ performance improvements requiring only 78% and 12% of the cache memory size for the Z-order scheme respectively at full and half resolution.

We plan to further investigate how this scheme affects memory cache efficiency at upper level (L1 or L2 cache) in the memory hierarchy. Our future plan also includes

application to 4-Dimensional data for efficient out-of-core time-varying volume visualization.

REFERENCES

[1] *The Visible Human project*, National Library of Medicine, National Institutes of Health, http://www.nlm.nih.gov/research/visible/visiblehuman.html.

[2] H. Samet, *The design and analysis of spatial data structures*. Addison-Wesley, 1990.

[3] J. Vitter, "External memory algorithms and data structures: Dealing with massive data." *ACM Computing Surveys*, March 2000.

[4] C. Hansen and C. Johnson, *The visualization handbook*. Elsevier Butterworth-Heinemann, 2005.

[5] M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proceedings of the 8th conference on Visualization*, 1997.

[6] P. M. Sutton and C. D. Hansen, "Accelerated isosurface extraction in time-varying fields," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 2, pp. 98–107, Apr 2000.

[7] Y.-J. Chiang, "Out-of-core isosurface extraction of time-varying fields over irregular grids," in *Proceedings of IEEE Visualization*, 2003, pp. 29–36.

[8] Q. Shi and J. JaJa, "Isosurface extraction and spatial filtering using persistent octree," in *IEEE Visualization*, 2006.

[9] H. Shen, L. Chiang, and K. Ma, "A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree," *Proceedings of the conference on Visualization'99: celebrating ten years*, pp. 371–377, 1999.

[10] P. Leutenegger and K. Ma, "Fast retrieval of disk resident unstructured volume data for visualization," *External Memory Algorithms and Visualization*, vol. 50, 1999.

[11] R. Farias and C. Silva, "Out-of-core rendering of large unstructured grids," *IEEE Computer Graphics and Applications*, vol. 21, no. 4, pp. 42–50, 2001.

[12] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom, "Out-of-core algorithms for scientific visualization and computer graphics," *IEEE Visualization Course Notes*, 2002.

[13] J. Orenstein and T. Merrett, "A class of data structures for associative searching," in *Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, April 1984, pp. 181–190.

[14] D. Hilbert, "Ueber stetige abbidung einer linie auf ein flachenstuck," *Mathematische Annalen*, vol. 38, pp. 459–460, 1891.

[15] J. Lawder, "The application of space-filling curves to the storage and retrieval of multi-dimensional data," Ph.D. dissertation, University of London, 2000.

[16] V. Pascucci and R. Frank, "Global Static Indexing for Real-Time Exploration of Very Large Regular Grids," in *Supercomputing, ACM/IEEE 2001 Conference*, 2001, pp. 45–45.