Global Static Indexing for Real-time Exploration of Very Large Regular Grids

Valerio Pascucci
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
pascucci@llnl.gov

Randall J. Frank
Livermore Computing
Lawrence Livermore National Laboratory
rjfrank@llnl.gov

ABSTRACT

In this paper we introduce a new indexing scheme for progressive traversal and visualization of large regular grids. We demonstrate the potential of our approach by providing a tool that displays at interactive rates planar slices of scalar field data with very modest computing resources. We obtain unprecedented results both in terms of absolute performance and, more importantly, in terms of scalability. On a laptop computer we provide real time interaction with a 2048^3 grid (8 Giga-nodes) using only 20MB of memory. On an SGI Onyx we slice interactively an 8192^3 grid (½ tera-nodes) using only 60MB of memory. The scheme relies simply on the determination of an appropriate reordering of the rectilinear grid data and a progressive construction of the output slice. The reordering minimizes the amount of I/O performed during the out-of-core computation. The progressive and asynchronous computation of the output provides flexible quality/speed tradeoffs and a time-critical and interruptible user interface.

1. INTRODUCTION

The real time processing of very large volumetric meshes introduces specific algorithmic challenges due to the impossibility of fitting the input data in the main memory of a computer. The basic assumption (RAM computational model) of uniform-constant-time access to each memory location is not valid because part of the data is stored out-of-core or in external memory. The performance of most algorithms does not scale well in the transition from the in-core to the out-of-core processing conditions. The performance degradation is due to the high frequency of I/O operations that may start dominating the overall running time.

Out-of-core computing [29] addresses specifically the issues of algorithm redesign and data layout restructuring to enable data access patterns with minimal performance degradation in out-of-core processing. Results in this area are also valuable in parallel and distributed computing where one has to deal with the similar issue of balancing processing time with data migration time.

In this paper we introduce a new storage layout for rectilinear grids of data that minimizes the amount of disk access necessary during a progressive traversal. The layout is coupled with a simple mapping of the 3D index (i, j, k) of an element in the grid to its 1D index I on disk. This new mapping improves on the approach introduced in [25] by using a conceptual 2^n tree decomposition instead of a binary tree.

The scheme has three key features that make it particularly attractive. First the order of the data is independent of the out-of-core blocking factor so that its use in different settings (e.g. local disk access or network transmission) does not require large data reorganization. Secondly the conversion from the standard Z-order indexing to the new index can be implemented with a simple sequence of bit-string manipulations making it appealing for possible hardware implementations. Third, there is no data replication. This is especially desirable when the data is accessed through slow connections and avoids performance degradation when the data is dynamically modified.

We use this approach to optimize progressive visualization algorithms where the input grid is traversed like a hierarchical structure (from the coarse level to the fine level) while displaying a continuously improved image of the output. In this paper we focus our attention to the case of progressive computation of planar slices with general orientation. The scheme achieves interactive performance for progressive slicing of extremely large datasets using moderate resources. On a laptop computer we provide real time interaction with a grid of 2048^3 using only 20MB of cache memory. Until recently real time navigation with this dataset [21] was only possible on a large multiprocessor systems, limiting interaction to the computation of orthogonal slices and requiring the duplication of the data for each slicing direction. With the new approach, data replication is not necessary and on the same multiprocessor system we can slice interactively at any orientation a dataset of 8192^3 resolution (½ tera-nodes grid) using only 60MB of cache memory and four threads.

2. PREVIOUS WORK

Interest in external memory algorithms [29], also known as out-of-core algorithms, has been increasing in recent years in the computer science community since they address systematically the problem of non-uniform memory structure of modern computers (fast cache, main memory, hard disk, etc). This issue is particularly important when dealing with large data-structures that do not fit in the main memory of a single computer since the access time to each memory unit is dependent on its location. New algorithmic techniques and analysis tools have been developed to address this problem for example in the case of

SC2001 November 2001, Denver © 2001 ACM 1-58113-293-X/01/0011 $5.00

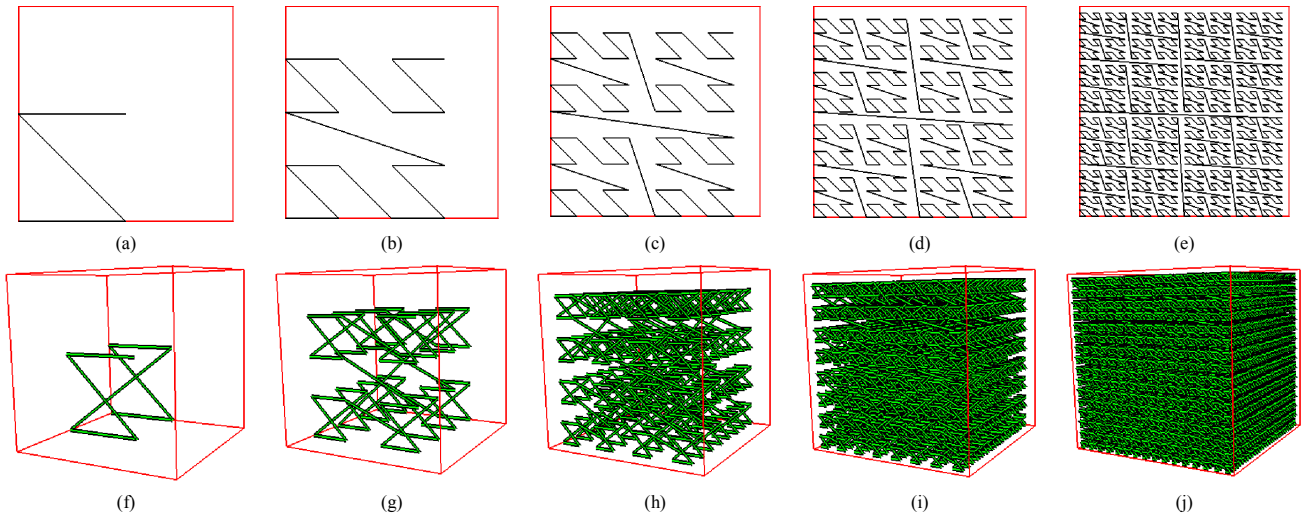


Figure 1: (a-e) The first five levels of resolution of the 2D Lebesgue's space filling curve. (f-j) The first five levels of resolution of the 3D Lebesgue's space filling curve.

geometric algorithms [1][2][13][20] or scientific visualization [4][10]. Closely related issues emerge in the area of parallel and distributed computing where remote data transfer can become the primary bottleneck in the computation. In this context space filling curves are often used as a tool to determine very quickly data distribution layouts that guarantee good geometric locality [14][22][24]. Space filling curves [27] have been also used in a wide variety of applications [3] both because of their hierarchical fractal structure as well as for their spatial locality properties. The most popular is the Hilbert curve [15], which guarantees the best geometric locality properties [23]. The pseudo-Hilbert scanning order [7][8][16] generalizes the scheme to rectilinear grids that have different number of samples along each coordinate axis.

Recently Lawder [17][18] explored the use of different kinds of space filling curves to develop indexing schemes for data storage layout and fast retrieval in multi-dimensional databases. Balmelli at al. [5][6] use the Z-order space filling curve to navigate efficiently a quad-tree data-structure without using pointers. They use simple, closed formulas for computing neighboring relations and nearest common ancestors between nodes to allow fast generation of adaptive edge-bisection triangulations. They improve on the basic data-structure already used for terrain visualization [11][19] or adaptive mesh refinement [26]. The use of the Z-order space filling curve for traversal of quadtrees [28] (also called Morton-order) and has been also proven useful in the speedup of matrix operations allowing them to make better use of the memory cache hierarchies [9][12][29].

In the approach proposed here a new data layout is used to allow efficient progressive access to volumetric information stored in external memory. This is achieved by combining interleaved storage of the levels in the data hierarchy while maintaining geometric proximity within each level of resolution. One main advantage is that the resulting data layout is independent of the particular adaptive traversal of the data. Moreover the same data layout can be used with different blocking factors making it beneficial throughout the entire memory hierarchy.

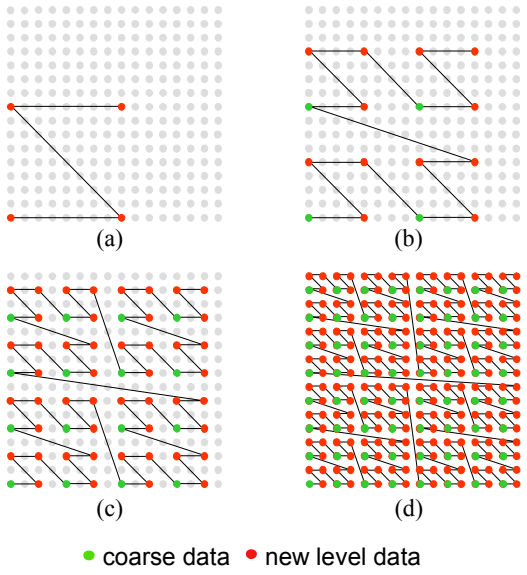


Figure 2: The sequence of Z-order space filling curves used to define a hierarchical sub-sampling stricture over a 2D rectilinear grid. At each refinement of the Z curve the green vertices represent the coarse data, while the red vertices represent the finer resolution information. Samples are stored in the order of the curve traversal, with only the new level data stored for each level of the hierarchy.

3. GENERAL FRAMEWORK

This section discusses a general framework for the definition of hierarchical indexing schemes yielding efficient external memory processing performance. The simplest realization of this scheme is based on mere sub-sampling the input data. For example we will show in the following sections how the Lebesgue space-filling curve of Figure 1a-e can be used as in Figure 2 to define a hierarchical sub-sampling of a rectilinear

grid. More sophisticated schemes can be realized within this framework but will not discussed further here.

Consider a set S of n elements decomposed into a hierarchy H of k levels of resolution $H = \{S_0, S_1, \ldots, S_{k-1}\}$ such that:

$$S_0 \subset S_1 \subset \ldots \subset S_{k-1} = S$$

where S_i is said to be coarser than S_j iff $i < j$. The order of the elements in S is defined by the cardinality function $I : S \rightarrow \{0 \ldots n\text{-}1\}$. This means that the following identity always holds:

$$S[I(s)] \equiv s$$

where the square brackets are used to index an element in a set.

We can define a derived sequence H' of sets S'_i as follows:

$$S'_i = S_i \setminus S_{i-1} \qquad i = 0, \ldots, k\text{-}1$$

Where formally $S_{-1} = \varnothing$. The sequence $H' = \{ S'_0, S'_1, \ldots, S'_{k-1}\}$ is a partitioning of S. A derived cardinality function $I' : S \rightarrow \{0 \ldots n\text{-}1\}$ can be defined on the basis of the following properties:

$$\forall s, t \in S'_i : I'(s) < I'(t) \Leftrightarrow I(s) < I(t);$$
$$\forall s \in S'_i, \forall t \in S'_j : i < j \Rightarrow I'(s) < I'(t).$$

If the original function I has strong locality properties when restricted to any level of resolution S_i then the cardinality function I' generates the desired global index for hierarchical and out-of-core traversal.

The construction of the function can be achieved in the following manner: (i) determine the number of elements in each derived set S'_i and (ii) determine a cardinality function $I''_i = I' | S'_j$ restriction of I' to each set S'_j. In particular if c_i is the number of elements of S'_i one can predetermine the starting index of the elements in a given level of resolution by building the sequence of constants $C_0, \ldots, C_{k-1}$ with

$$C_i = \sum_{j=0}^{i-1} c_j \qquad (1)$$

Secondly one needs to determine a set of local cardinality functions $I''_i : S'_j \rightarrow \{0 \ldots c_i\text{-}1\}$ so that

$$\forall s \in S'_i : I'(s) = C_i + I''_i(s) \qquad (2)$$

The computation of the constants C_i can be performed in a pre-processing stage so that the computation of I' is reduced to the following two steps:

(1) Given s, determine its level of resolution i (that is the i such that $S \in S'_i$);

(2) Compute $I''_i(s)$ and add it to C_i

These two steps need to be performed very efficiently because they are going to be executed repeatedly at run time. The following section reports a practical realization of this scheme for rectilinear cube grids in any dimension.

4. 2^n TREE AND THE LEBESQUE CURVE

Here we detail the derivation from the Z-order space filling curve the local cardinality functions I''_i for a binary tree hierarchy in any dimension.

4.1 Indexing the Lebesgue Curve

The Lebesgue space filling curve, also called Z-order space filling curve for its shape in the 2D case, is depicted in Figure 1. In the 2D case the curve can be defined inductively by a base Z shape of size 1 (Figure 1a) whose vertices are replaced each by a Z shape of size ½. The vertices obtained are then replaced by Z shapes of size ¼ (Figure 1c) and so on. In general the i^{th} level of resolution is defined as the curve obtained by replacing the vertices of the $(i-1)^{th}$ level of resolution with Z shapes of size $(1/2^i)$. The 3D version of this space filling curve has the same hierarchical structure with the only difference that the basic Z shape is replaced by a connected pair of Z shapes lying on the opposite faces of a cube as shown in f. Figure 1f-j shows five successive refinements of the 3D Lebesgue space filling curve. The d-dimensional version of the space filling curve has also the same hierarchical structure where the basic shape (the Z of the 2D case) is defined as a connected pair of $(d-1)$-dimensional basic shapes lying on the opposite faces of a d-dimensional cube.

The property that makes the Lebesgue's space filling curve particularly attractive is the easy conversion from the d indices of d-dimensional matrix to the 1D index along the curve. If one element e has d-dimensional reference $(i_1, \ldots, i_d)$ its 1D reference is built by interleaving the bits of the binary representations of the indices $i_1, \ldots, i_n$. In particular if i_j is represented by the string of h bits "$b^1_j\ b^2_j \ldots b^h_j$" with $j = 1, \ldots, d$) then the 1D reference of e is represented the string of hd bits I = "$b^1_1\ b^1_2 \ldots b^1_d\ b^2_1\ b^2_2 \ldots b^2_d \ldots b^h_1\ b^h_2 \ldots b^h_d$". Figure 3 illustrates this interleaving scheme in the 3D case.

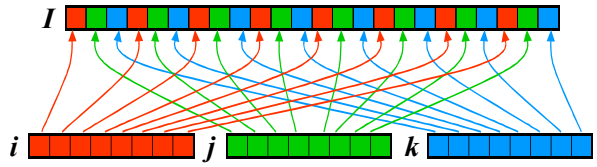


Figure 3: Construction of the 1D index for the Lebesgue's Z-order space filling curve. In the 3D case the original index is a set of three bit-strings *(i,j,k)*. The 1D index *I* is formed by interleaving the bits of *i,j* and *k* into a single bit-string.

4.2 Index Remapping

The cardinality function discussed in Section 3 for the case of a quad-tree yields the index remapping shown in Table 1 for the first three levels.

The structure of the 2^l - tree defined on the Z-order space filling curve allows to determine easily the three elements that are necessary for the computation of the cardinality which are: (i) the level i of an element, (ii) the constants C_i of equation (1) and (iii) the local indices I''_i. Following the notation of Section 3 we have:

i - if the 2^l-tree hierarchy has k levels then the element of Z-order index j belongs to the level k - h where h is the number of trailing zeros in the binary representation of j divided by l:

$$i = k - \lfloor h \rfloor$$

Table 1: Structure of the hierarchical indexing scheme for a quad-tree combined with the order defined by the Lebesgue space filling curve.

Level	0	1			2											
Z-order index (1 level)	0															
Z-order index (2 levels)	0	1	2	3												
Z-order index (3 levels)	0	4	8	12	1	2	3	5	6	7	9	10	11	13	14	15
Hierarchical index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

C_i - the number of elements in the levels coarser than I, with $i > 0$, is

$$C_i = 2^{l(i-1)}$$

with $C_0 = 0$.

I''_i - if an element has index j and belongs to the set S''_i then $\lfloor j/2^{li} \rfloor$ has one of its last l bits different from 0. The local index is then:

$$I''_i(j) = \lfloor j/2^{li} \rfloor - \lfloor j/2^{l(i+1)} \rfloor - 1$$

The computation of the local index I''_i can be explained easily by looking at the bottom right part of Table 1 where the sequence on indices (1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15) needs to be remapped to the local index (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11). Note how each original sequence of three consecutive numbers needs to be decreased by the same amount. From the sequence (1,2,3) one has to subtract 1 to get (0, 1, 2). From (5, 6, 7) one has to subtract 2 to get (3, 4, 5) and so on. In general the subtrahend is 1 plus the index divided by four.

The computation of the final index $I'(s) = C_i + I''_i(s)$ is then

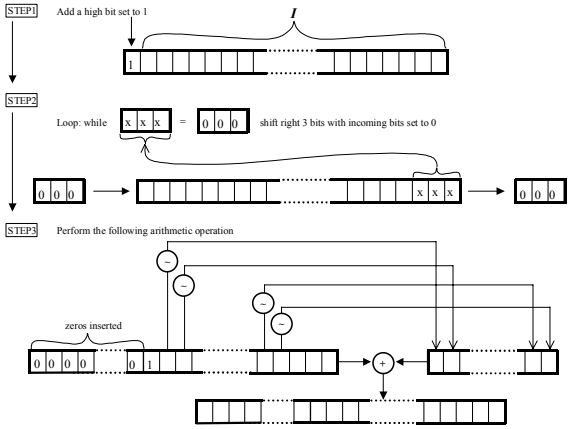


Figure 4: Diagram of the algorithm for index remapping for blocks that scale with a factor of 8 (three bits).

performed with the algorithm shown in Figure 4. Note how the term sum $C_i + I''_i(s)$ is computed directly by shifting $I(s)$ to the right and adding the complement of its high bits. This can be done because $C_i = 2^{l(i-1)}$ and hence $C_i - \lfloor j/2^{l(i+1)} \rfloor - 1$ can be obtained directly by complementing the rightmost $l(k - i - 1)$ bits of $\lfloor j/2^{l(i+1)} \rfloor$.

Figure 5 illustrates the data layout when the elements of a 2D matrix are reordered following the index I'. The general pattern that emerges is that the decomposition of the 1D array of data into a sequence of blocks of the same size (the first is slightly larger) corresponds to data that is distributed increasingly locally on the 2D matrix.

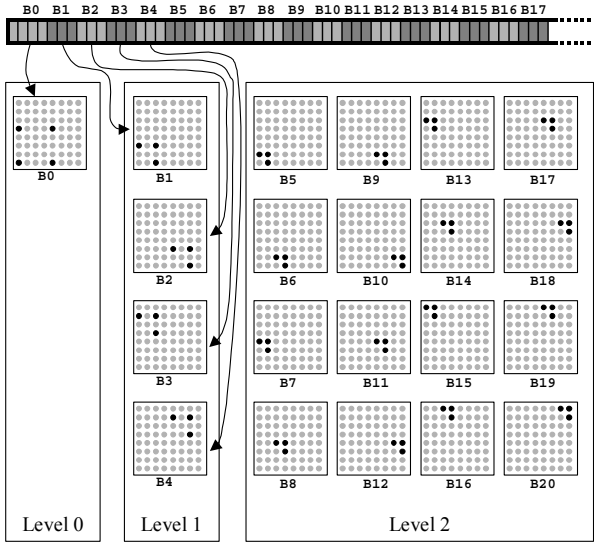
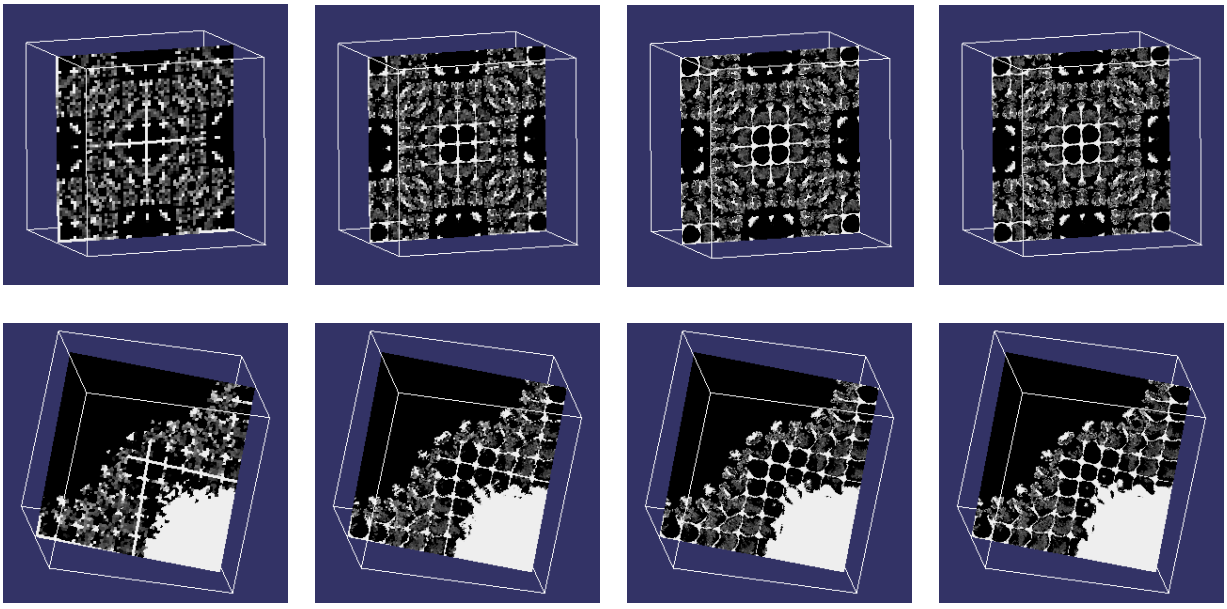


Figure 5: Data layout obtained for a 2D matrix. The 1D array at the top of the diagram represents the disk distribution of the data. Each of the consecutive blocks in the 1D array corresponds to data of progressively finer resolution in the 2D matrix, distributed in an increasingly

5. PROGRESSIVE SLICING

To demonstrate the utility of this indexing scheme, an implementation of the indexing scheme for the computation and display of arbitrary slices based on standard operating system I/O primitives was developed for Unix and Windows. The data access model consists of a fixed size memory cache and a compressed disk file format with associated meta-data. A planar slice through the dataset is realized as a set of point samples on at 2D grid orientated with arbitrary attitude in the data volume. The input to the I/O layer is this set of points and their associated level (resolution) in the index hierarchy. Individual sample points are converted into a data index using the hierarchical Z order space-filling curve. This index is converted to a virtual block number and a local index within the block by simple division. The block number is queried in the cache. If the block is in the cache, the sample for the point is accessed and returned, otherwise, an asynchronous I/O operation for that block is added to an I/O queue and the point marked as pending. Point processing continues until all points have been resolved (including pending points) or the system has exceeded a predetermined processing time limit. The block cache is filled



(sampling rate=1/32) (sampling rate=1/16) (sampling rate=1/8) (sampling rate=1/4)

Figure 6: Progressive refinement of two slices of one timestep of the 2048x2048x1920 dataset from [21]. The images in the left column are rendered at the coarsest resolution using one sample every thirtytwo along each axis of the input grid. Each row shows the progressive sequence of textures obtained for each slice by increasing the sampling rate up to 1/4. Note how the detail is increased with the sampling rate. (top row) Slice perpendicular to the *z* axis. (bottom row) Slice at an arbitrary orientation.

asynchronously by I/O threads, which read compressed blocks from disk, decompress them into the cache, and resolve any sample points pending on that cache operation.

An interface thread determines the orientation and position of the current slice plane and sends planar requests to the I/O layer. As data values return from the I/O layer the interface thread fills in the output texture while the plane is being displayed asynchronously. This simple mechanism produces instant coarse resolution and progressive improvement of the slice image (see Figure 6).

The current implementation does not include speculative prefetching of the data or other mechanisms that pipeline the cost of the I/O performed with the exception of threaded block decompression. The timings in the charts include both the complete disk access time and data decompression time for each frame. In this way we are able to evaluate of the total amount of resources used and perform fair comparisons among several alternative indexing schemes. The blocking factor for the data was selected arbitrarily to be 64KB and compression was provided by zlib. Further improvement of the interactivity may be achieved utilizing a more efficient data compression methods and data-prefetching.

We compare the performance of the same progressive slicing scheme for four different data layouts: (Array) storage of the data as a standard row major array, (brick blocking) decomposition of the data in cubes of size equal to the disk pages, (1-bit hierarchical Z-order) introduced in [25] and (3-bit hierarchical Z order) introduced in the previous section. In the graphs the rotation tests report average timings for rotating slices over each primary axis, while the translation tests report the average time for translating orthographic slices along each axis. The dataset is one timestep of the PPM dataset [21] shown

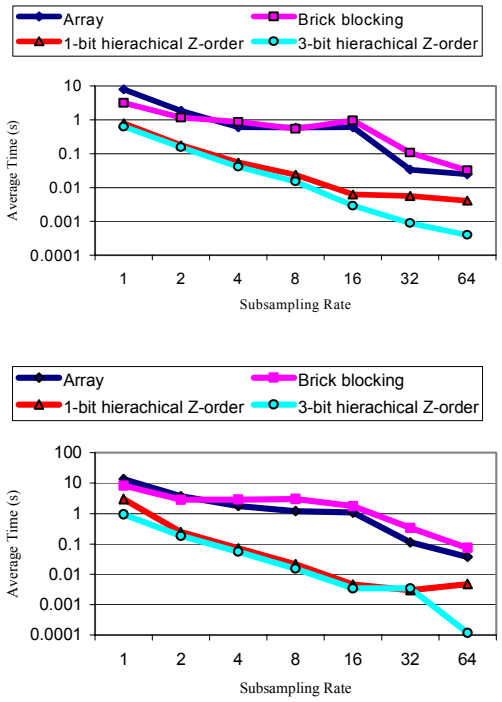


Figure 7: Average 512^2 slice computation times for a 2048^3 grid on a Linux laptop (500Mhz Pentium III, 20MB memory cache). (top) Slicing times obtained averaging the slice computation time over a sequence of parallel slices. (bottom) Slicing times obtained averaging the slice computation time over a sequence of slices rotate around a common axis.

in Figure 6. The timestep is a 2048x2048x1920 array of 8bit intensity values. Note that the timings are reported in logarithmic scale. Generally our new 3-bit hierarchical Z-order outperforms all other schemes. The 3-bit hierarchical Z order can be twice as fast as the 1-bit hierarchical Z order and orders of magnitude faster than the array and brick decomposition schemes.

Figure 7 shows the timings obtained on a Linux laptop for progressive slicing of the 2048x2048x1920 grid. The amount of cache memory used to achieve this result is only of 20MB.

Figure 8 shows the timing obtained on an SGI Onyx2 for progressive slicing of an artificial 8192^3 dataset obtained by replicating 64 times (4x4x4) one timestep of the PPM dataset. 60MB of RAM and four threads were used in generating this result. The scheme appears to scale very well with the size of the input grid as witnessed by the relatively modest increase in computational resources necessary to scale to a dataset 64 times larger. More problematic is the scaling with the size of the output texture. The results presented in the previous graphs are based on output textures of 512^2 resolution. This resolution many not be sufficient for newer high-pixel count displays.

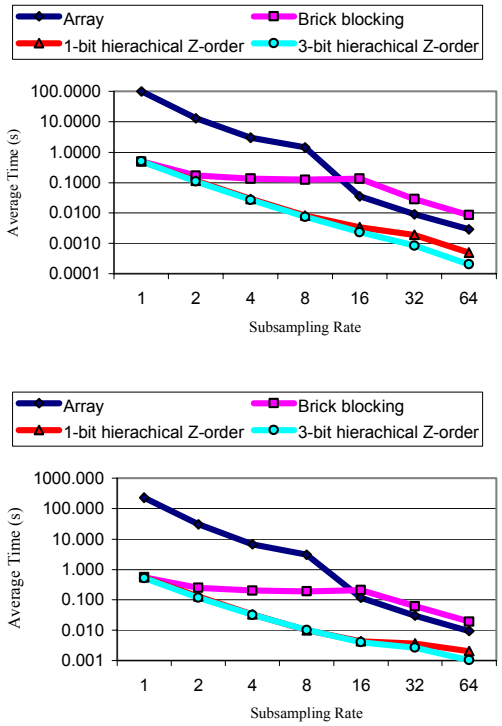


Figure 8: Average 512^2 slice computation times for a 2048^3 grid on an SGI Onyx (300Mhz R12000, 60MB memory cache). (top) Slicing times obtained averaging the slice computation time over a sequence of parallel slices. (bottom) Slicing times obtained averaging the slice computation time over a sequence of slices rotate around a common axis.

Figure 9 shows the timing for an output texture of 2048^2 resolution, which may be more appropriate for such displays. Performance begins to decline as the higher output resolution exposes inefficiencies in our caching and queuing implementations. For example, a much larger memory cache is necessary to avoid trashing as the output matrix size increases. Fortunately only a moderate amount processing power is used by the scheme, so that we can use object space parallelism to improve performance. In particular we can maintain the same level of interactivity shown in Figure 8 by dividing the texture in 16 tiles of 512^2 resolution and use them to compose a high resolution output texture at real time rates. It may also be possible to improve the performance of the serial implementation by enhancing our cache implementation. The need for increased cache memory may be minimized by traversing the 2D grid along a space filling curve as well.

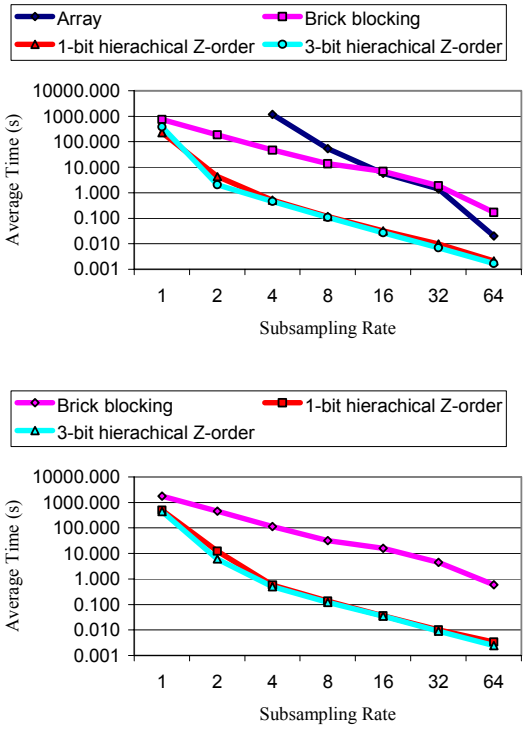


Figure 9: Average 2048^2 slice computation times for a 8192^3 grid on an SGI Onyx (300Mhz R12000, 60MB memory cache). (top) Slicing times obtained averaging the slice computation time over a sequence of parallel slices. (bottom) Slicing times obtained averaging the slice computation time over a sequence of slices rotate around a common axis.

6. CONCLUSIONS

This paper introduces a new indexing and data layout scheme that is useful for out-of-core hierarchical traversal of large datasets. Practical tests for the case of progressive slicing of rectilinear grids demonstrate the performance improvements that can be achieved with this approach. For example we can translate and rotate planar slices of an 8192^3 grid achieving real-time interaction rates. In the near future this scheme will be used as the basis for out-of-core volume visualization, computation of isocontours and navigation of large terrains.

Future directions being considered include the combination with wavelet compression schemes, the extension to general

rectangular grids, distributed memory implementations and application to non-rectilinear hierarchies.

7. ACKNOWLEDGMENTS

We would like to thank to a number of people who contributed to this project, including Mark Duchaineau and Sean Ahern at LLNL. The example dataset appears courtesy of Art Mirin of LLNL.

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-JC-144754). It was supported in part by the Accelerated Strategic Computing Initiative and the Visual Interactive Environment for Weapons Simulations (VIEWS) program.

8. REFERENCES

[1] Abello, J., and Vitter, J.S., (eds.). *External Memory Algorithms and Visualizatio*n. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[2] Arge, L., and Miltersen, P.B., On showing lower bounds for external-memory computational geometry problems. In Abello, J., and Vitter, J.S., editors, *External Memory Algorithms and Visualizatio*n, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[3] Asano, T., Ranjan, D., Roos, T., and Welzl, E., Space filling curves and their use in the design of geometric data structures. *Lecture Notes in Computer Scienc*e, 911:36–44, 1995.

[4] Bajaj, C.L., Pascucci, V., Thompson, D., and Zhang, X.Y., Parallel accelerated isocontouring for out-of-core visualization. In Spencer, S.N., editor, *Proceedings of the 1999 IEEE Parallel Visualization and Graphics Symposium (PVGS'99*), pages 97–104, N.Y., October 25–26 1999. ACM Siggraph.

[5] Balmelli, L., Kovačević, J., and Vetterli, M., Quadtree for embedded surface visualization: Constraints and efficient data structures. In *IEEE International Conference on Image Processing (ICIP*), Kobe Japan, October 1999.

[6] Balmelli, L., Kovačević, J., and Vetterli, M., Solving the coplanarity problem of regular embedded triangulations. In *Proceedings of the Workshop on Vision, Modeling and Visualizatio*n, November 1999.

[7] Bandou, Y., and Kamata, S.I., An address generator for a 3-dimensional pseudo-hilbert scan in a cuboid region. In *International Conference on Image Processing, ICIP99*, volume I, 1999.

[8] Bandou, Y., and Kamata, S.I., An address generator for an n-dimensional pseudo-hilbert scan in a hyper-rectangular parallelepiped region. In *International Conference on Image Processing, ICIP 2000*, 2000. to appear.

[9] Chatterjee, S., Lebeck, A.R., Patnala, P.K., and Thottethodi, M., Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architecture*s, pages 222–231, Saint-Malo, France, June 27–30, 1999. SIGACT/SIGARCH and EATCS.

[10] Chiang, Y.J., and Silva, C.T., I/O optimal isosurface extraction. In Yagel, R., and Hagen, H., editors, *IEEE Visualization´ 9*7, pages 293–300. IEEE, November 1997.

[11] Duchaineau, M.A., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., and Mineev-Weinstein, M.B., Roaming terrain: Real-time optimally adapting meshes. *IEEE Visualization '9*7, pages 81–88, November 1997.

[12] Frens, J.D., and Wise, D.S., Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. *ACM SIGPLAN Notice*s, 32(7):206–216, July 1997.

[13] Goodrich, M.T., Tsay, J.J., Vengroff, D.E., and Vitter, J.S., External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS '93*), Palo Alto, CA, November 1993.

[14] Griebel, M., and Zumbusch, G.W., Parallel multigrid in an adaptive pde solver based on hashing and space-filling curves. 25:827:843, 1999.

[15] Hilbert, D., Über die stetige abbildung einer linie auf ein flachenstück. *Mathematische Annale*n, 38:459–460, 1891.

[16] Kamata, S.I., and Bandou, Y., An address generator of a pseudo-hilbert scan in a rectangle region. In *International Conference on Image Processing, ICIP9*7, volume I, pages 707–714, 1997.

[17] Lawder, J.K., *The Application of Space-filling Curves to the Storage and Retrieval of Multi-Dimensional Data*. PhD thesis, School of Computer Science and Information Systems, Birkbeck College, University of London, 2000.

[18] Lawder, J.K., and King, P.J.H., Using space-filling curves for multi-dimensional indexing. In Brian Lings and Keith Jeffery, editors, *proceedings of the 17th British National Conference on Databases (BNCOD 17*), volume 1832 of *Lecture Notes in Computer Scienc*e, pages 20–35. Springer Verlag, July 2000.

[19] Lindstrom, P., Koller, D., Ribarsky, W., Hughes, L.F., Faust, N., and Turner, G., Real-time, continuous level of detail rendering of height fields. *Proceedings of SIGGRAPH 9*6, pages 109–118, August 1996.

[20] Matias, Y., Segal, E., and Vitter, J.S., Efficient bundle sorting. In *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA '00*), January 2000.

[21] Mirin, A., Performance of large-scale scientific applications on the IBM ASCI blue-pacific system. In *Ninth SIAM Conf. of Parallel Processing for Scientific Computin*g, Philadelphia, Mar 1999. SIAM. CD-ROM.

[22] Niedermeier, R., Reinhardt, K., and Sanders, P., Towards optimal locality in meshindexings, 1997.

[23] Niedermeier, R., and Sanders, P., On the manhattan-distance between points on space-filling mesh-indexings. Technical Report iratr-1996-18, Universität Karlsruhe, Informatik für Ingenieure und Naturwissenschaftler, 1996.

[24] Parashar, M., Browne, J.C., Edwards, C., and Klimkowski, K., A common data management infrastructure for adaptive algorithms for pde solutions. In *SuperComputing 9*7, 1997.

[25] Pascucci, V., and Frank, R.J., Hierarchical indexing for out-of-core access to multi-resolution data. Technical Report UCRL-JC-140581, Lawrence Livermore National Laboratory, 2001. A preliminary version was presented at the Lake Tahoe Workshop NSF/DOE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization.

[26] Rivara, M.C., Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International Journal for Numerical Methods in Engineerin*g, 20:745–756, 1984.

[27] Sagan, H., *Space-Filling Curve*s. Springer-Verlag, New York, NY, 1994.

[28] Samet, H., *Applications of Spatial Data Structure*s. Addison-Wesley, Reading, Mass., 1990. chapter on ray tracing and efficiency, also discusses radiosity.

[29] Vitter, J.S., External memory algorithms and data structures: Dealing with massive data. *ACM Computing Survey*s, March 2000.

[30] Wise, D.S., Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In Euro-Par 2000 – Parallel Processing, volume 1900 of Lecture Notes in Computer Science, pages 774–784. Springer, August 2000.