1 Hierarchical Indexing for Out-of-Core Access to Multi-Resolution Data

Valerio Pascucci and Randall J. Frank

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.

Summary. Increases in the number and size of volumetric meshes have popularized the use of hierarchical multi-resolution representations for visualization. A key component of these schemes has become the adaptive traversal of hierarchical data-structures to build, in real time, approximate representations of the input geometry for rendering. For very large datasets this process must be performed out-of-core. This paper introduces a new global indexing scheme that accelerates adaptive traversal of geometric data represented with binary trees by improving the locality of hierarchical/spatial data access. Such improvements play a critical role in the enabling of effective out-of-core processing.

Three features make the scheme particularly attractive: (i) the data layout is independent of the external memory device blocking factor, (ii) the computation of the index for rectilinear grids is implemented with simple bit address manipulations and (iii) the data is not replicated, avoiding performance penalties for dynamically modified data.

The effectiveness of the approach was tested with the fundamental visualization technique of rendering arbitrary planar slices. Performance comparisons with alternative indexing approaches confirm the advantages predicted by the analysis of the scheme.

1.1 Introduction

The real time processing of very large volumetric meshes introduces unique algorithmic challenges due to the impossibility of fitting the 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 many algorithms does not scale well in the transition from the in-core to the out-of-core processing conditions. This performance degradation is due to the high frequency of I/O operations that start dominating the overall running time (trashing).

Out-of-core computing [22] addresses specifically the issues of algorithm redesign and data layout restructuring, necessary to enable data access patterns with minimal out-of-core processing performance degradation. Research in this area is also valuable in parallel and distributed computing, where one has to deal with the similar issue of balancing processing time with data migration time.

The solution of the out-of-core processing problem is typically divided into two parts:

(i) algorithm analysis to understand its data access patterns and, when possible, redesign to maximize their locality;

(ii) storage of the data in secondary memory with a layout consistent with the access patterns of the algorithm, amortizing the cost individual I/O operations over several memory access operations.

In the case of hierarchical visualization algorithms for volumetric data, the 3D input hierarchy is traversed to build derived geometric models with adaptive levels of detail. The

shape of the output models are then modified dynamically with incremental updates of their level of detail. The parameters that govern this continuous modification of the output geometry are dependent on runtime user interaction, making it impossible to determine, *a priori*, what levels of detail will be constructed. For example, parameters can be external, such as the viewpoint of the current display window or internal, such as the isovalue of an isocontour or the position of an orthogonal slice. The general structure of the access pattern can be summarized into two main points: (i) the input hierarchy is traversed coarse to fine, level by level so that the data in the same level of resolution is accessed at the same time and (ii) within each level of resolution the data is mainly traversed coherently in regions that are in close geometric proximity.

In this paper we introduce a new static indexing scheme that induces a data layout satisfying both requirements (i) and (ii) for the hierarchical traversal of n-dimensional regular grids. 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 transmission through a network) does not require any large data reorganization. Second, conversion from the standard Z-order indexing to the new indexing scheme can be implemented with a simple sequence of bit-string manipulations making it appealing for a possible hardware implementation. Third, there is no data replication, avoiding any performance penalty for dynamic updates or any inflated storage typical of most hierarchical and out-of-core schemes.

Beyond the theoretical interest in developing hierarchical indexing schemes for n-dimensional space filling curves, our approach targets practical applications in out-of-core visualization algorithms. In this paper, we report algorithmic analysis and experimental results for the case of slicing large volumetric datasets.

The remainder of this paper is organized as follows. Section 1.2 discusses briefly previous work in related areas. Section 1.3 introduces the general framework for the computation of the new indexing scheme. Section 1.4 discusses the implementation of the approach for binary tree hierarchies. Section 1.5 analyzes the application of the scheme for progressive computation of orthogonal slices reporting experimental timings for memory mapped files. Section 1.6 presents the structure of the I/O system and practical results obtained with compressed data. Concluding remarks and future directions are discussed in section 1.7.

1.2 Related Previous Work

External memory algorithms [22], also known as out-of-core algorithms, have been rising to the attention of the computer science community in recent years as they address, systematically, the problem of the non-uniform memory structure of modern computers (e.g. L1/L2 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 in the case of geometric algorithms [1,2,9,15] and scientific visualization [4,8]. Closely related issues emerge in the area of parallel and distributed computing where remote data transfer can become a 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 [10,18,20]. Space filling curves [21] have been also used in the past in a wide variety of applications [3] because of their hierarchical fractal structure as well as for their well

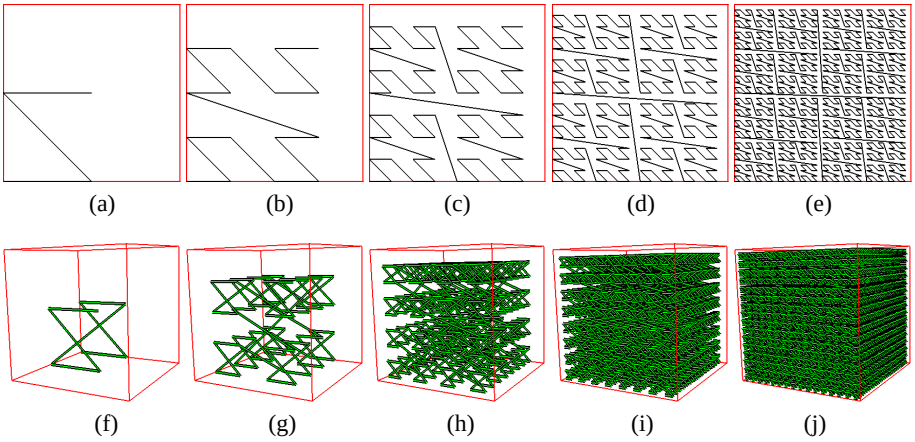


Fig. 1.1. (a-e) The first five levels of resolution of the 2D Lebesgue's space filling curve. (f-j) The first five levels of resolution of the 3D Lebesgue's space filling curve.

known spatial locality properties. The most popular is the Hilbert curve [11] which guarantees the best geometric locality properties [19]. The pseudo-Hilbert scanning order [7,6,12] generalizes the scheme to rectilinear grids that have different number of samples along each coordinate axis.

Recently Lawder [13,14] 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] use the Z-order (Lebesque) space filling curve to navigate efficiently a quad-tree data-structure without using pointers.

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 (multidimensional breadth first traversal). 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.

1.3 Hierarchical Subsampling Framework

This section discusses the general framework for the efficient definition of a hierarchy over the samples of a dataset.

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

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

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

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

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

One can define a derived sequence $\mathcal{H}'$ of sets S'_i as follow:

$$S'_i = S_i \backslash S_{i-1} \qquad i = 0, \dots, k-1$$

where formally $S_{-1} = \emptyset$. The sequence $\mathcal{H}' = \{S'_0, S'_1, \dots, S'_{k-1}\}$ is a partitioning of S. A derived cardinality function $I' : S \to \{0 \dots n-1\}$ can be defined on the basis of the following two 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 scheme has strong locality if elements with with close index are also close in geometric position. This locality properties are well studied in [17].

The construction of the function can be achieved in the following way: (i) determine the number of elements in each derived set S'_i and (ii) determine a cardinality function $I''_i = I'|_{S'_i}$ restriction of I' to each set S'_i. 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 $C_0, \dots, C_{k-1}$ with

$$C_i = \sum_{j=0}^{i-1} c_j. \tag{1.1}$$

Next, one must determine a set of local cardinality functions $I''_i : S'_i \to \{0 \dots c_i - 1\}$ so that:

$$\forall s \in S'_i : I'(s) = C_i + I''_i(s). \tag{1.2}$$

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

- given s determine its level of resolution i (that is the i such that $s \in S'_i$);
- compute $I''_i(s)$ and add it to C_i.

These two steps must be performed very efficiently as they will be executed repeatedly at run time. The following section reports a practical realization of this scheme for rectilinear cube grids in any dimension.

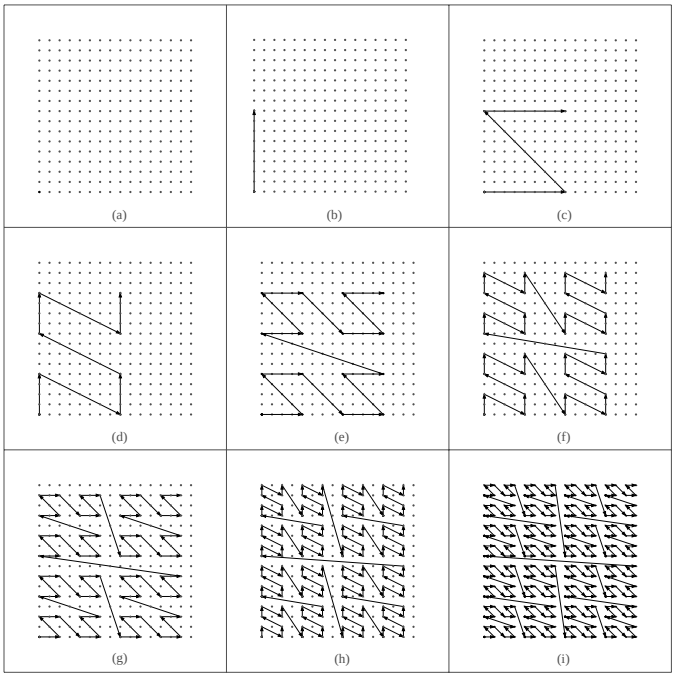


Fig. 1.2. The nine levels of resolution of the binary tree hierarchy defined by the 2D space filling curve applied on 16×16 rectilinear grid. The coarsest level of resolution (a) is a single point. The number of points that belong to the curve at any level of resolution (b-i) is double the number of points of the previous level.

1.4 Binary Trees And the Lebesgue Space Filling Curve

This section reports the details on how to derive from the Z-order space filling curve the local cardinality functions $I_i^{\prime\prime}$ for a binary tree hierarchy in any dimension and its remapping to the new index $I^{\prime}$.

1.4.1 Indexing the Lebesgue Space Filling Curve

The Lebesgue space filling curve, also called Z-order space filling curve for its shape in the 2D case, is depicted in figure 1.1(a-e). The Z-order space filling curve can be defined inductively by a base Z shape of size 1 (figure 1.1a) whose vertices are replaced each by a Z shape of size $\frac{1}{2}$ (figure 1.1b). The vertices obtained are then replaced by Z shapes of size $\frac{1}{4}$ (figure 1.1c) 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 $\frac{1}{2^i}$. 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 Figure 1.1(f). Figure 1.1(f-j) show 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 a 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 $i_1, \ldots, i_d$. In particular if i_j is represented by the string of h bits "$b_j^1 b_j^2 \cdots b_j^h$" (with $j = 1, \ldots, d$) then the 1D reference I of e is represented the string of hd bits $I =$ "$b_1^1 b_2^1 \cdots b_d^1 b_1^2 b_2^2 \cdots b_d^2 \cdots b_1^h b_2^h \cdots b_d^h$".

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

Table 1.1. Structure of the hierarchical indexing scheme for binary tree combined with the order defined by the Lebesgue space filling curve.

The 1D order can be structured in a binary tree by considering elements of level i, those that have the last i bits all equal to 0. This yields a hierarchy where each level of resolution has twice as many points as the previous level. From a geometric point of view this means that the density of the points in the d-dimensional grid is doubled alternatively along each coordinate axis. Figure 1.2 shows the binary hierarchy in the 2D case where the resolution of the space filing curve is doubled alternatively along the x and y axis. The coarsest level (a) is a single point, the second level (b) has two points , the third level (c) has four points (forming the Z shape) and so on.

1.4.2 Index Remapping

The cardinality function discussed in section 1.3 for a binary tree case has the structure shown in table 1.1. Note that this is a general structure suitable for out-of-core storage of static binary trees. It is independent of the dimension d of the grid of points or of the Z-order space filling curve.

The structure of the binary tree defined on the Z-order space filling curve allows one to easily determine the three elements necessary for the computation of the cardinality. They are: (i) the level i of an element, (ii) the constants C_i of equation (1.1) and (iii) the local indices I_i''.

i - if the binary tree hierarchy has k levels then the element of Z-order index j in the Z-order belongs to the level $k - h$, where h is the number of trailing zeros in the binary representation of j;

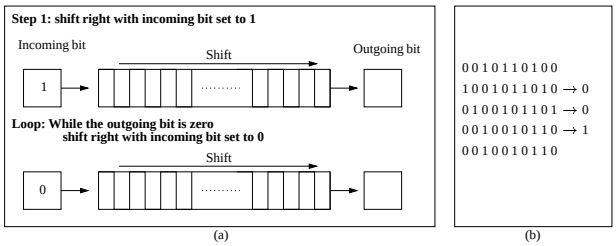


Fig. 1.3. (a) Diagram of the algorithm for index remapping from Z-order to the hierarchical out-of-core binary tree order. (b) Example of the sequence of shift operations necessary to remap an index. The top element is the original index the bottom is the output remapped index.

C_i - the total number of elements in the levels coarser than i, with $i > 0$, is $C_i = 2^{i-1}$ with $C_0 = 0$;

I_i'' - if an element has index j and belongs to the set S_i' then $\frac{j}{2^{k-i}}$ must be an odd number, by definition of i. Its local index is then:

$$I_i''(j) = \left\lfloor \frac{j}{2^{k-i+1}} \right\rfloor .$$

The computation of the local index I_i'' can be explained easily by looking at the bottom right part of table 1.1 where the sequence on indices (1 , 3 , 5 , 7 , 9 , 11 ,13 ,15) needs to be remapped to the local index (0 ,1 ,2 ,3 ,4 ,5 ,6 ,7). The original sequence is made of a consecutive series of odd numbers. A right shift of one bit (or rounded division by two) turns them into the desired sequence.

These three elements can be put together to build an efficient algorithm that computes the hierarchical index $I'(s) = C_i + I_i''(s)$ in the two steps shown in the diagram of Figure 1.3:

1. set to 1 the bit in position $k + 1$;
2. shift to the right until a 1 comes out of the bit-string.

Clearly this diagram could have a very simple and efficient hardware implementation. The software C++ version can be implemented as follows:

```
inline adhocidex remap(register adhocindex i){
  i |= last_bit_mask;  // set leftmost one
  i /= i&-i;           // remove trailing  zeros
  return (i>>1);       // remove rightmost one
}
```

This code would work only on machines with two's complement representation of numbers. In a more portable version one needs to replace `i /= i&-i` with `i /= i&((~i)+1)`.

Figure 1.4 shows the data layout obtained for a 2D matrix when its elements of are reordered following the index I'. The data is stored in this order and divided into blocks of constant size. The inverse image of such decomposition has the first block corresponding to the coarsest level of resolution of the data. The following blocks correspond to finer and finer resolution data that is distributed more and more locally.

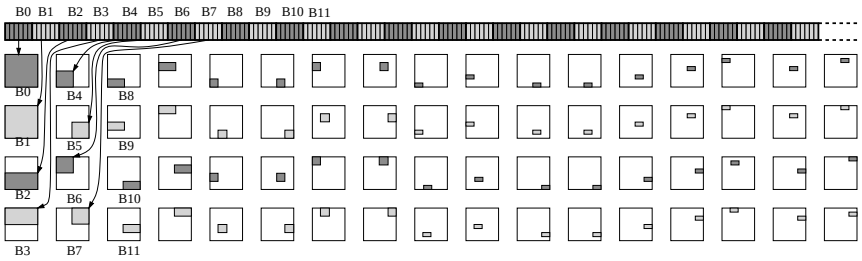


Fig. 1.4. Data layout obtained for a 2D matrix reorganized using the index I' (1D array at the top). The inverse image of the block decomposition of the 1D array is shown below. Each gray region shows where the block of data is distributed in the 2D array. In particular the first block is the set of coarsest levels of the data distributed uniformly on the 2D array.The next block is the next level of resolution still covering the entire matrix. The next two levels are finer data covering each half of the array. The subsequent blocks represent finer resolution data distributed with increasing locality in the 2D array.

1.5 Computation of Planar Slices

This section presents some experimental results based on the simple, fundamental visualization technique of computing orthogonal slices of a 3D rectilinear grid. The slices can be computed at different levels of resolution to allow real time user interactivity independent of the size of the dataset. The data layout proposed here is compared with the two most common array layouts: the standard row major structure and the $h \times h \times h$ brick decomposition of the data. Both practical performance tests and complexity analysis lead to the conclusion that the data layout proposed allows one to achieve substantial speedup both when used at coarse resolution and traversed in a progressive fashion. Moreover no significant performance penalty is observed if used directly at the highest level of resolution.

1.5.1 External Memory Analysis for Axis-Orthogonal Slices

The out-of-core analysis reports the number of data blocks transferred from disk under the assumption that each block of data of size b is transferred in one operation independently of how much data in the block is actually used. At fine resolution the simple row major array storage achieves the best and worst performances depending on the slicing direction. If the overall grid size is g and the size of the output is t, then the best slicing direction requires one to load $O(t/b)$ data blocks (which is optimal) but the worst possible direction requires one to load $O(t)$ blocks (for $b = \Omega(\sqrt[3]{g})$). In the case of simple $h \times h \times h$ data blocking (which has best performance for $h = \sqrt[3]{b}$) the number of blocks of data loaded at fine resolution are $O(\frac{t}{\sqrt[3]{b^2}})$. Note that this is much better than the previous case because the performance is close to (even if not) optimal, and independent of the particular slicing direction. For a subsampling rate of k the performance degrades to $O(\frac{tk^2}{\sqrt[3]{b^2}})$ for $k < \sqrt[3]{b}$. This means that at coarse subsampling, the performance goes down to $O(t)$. The advantage of the scheme proposed here is that independent of the level of subsampling, each block of data is used for a

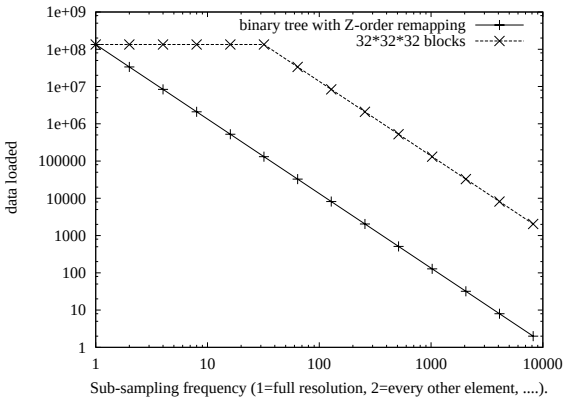


Fig. 1.5. Maximum data loaded from disk (vertical axis) per slice computed depending on the level of subsampling (horizontal axis) for an 8G dataset. (a) Comparison of the brick decomposition with the binary tree with Z-order remapping scheme proposed here. The values on the vertical axis are reported in logarithmic scale to highlight the difference in orders of magnitude at any level of resolution.

portion of $\sqrt[3]{b^2}$ so that, independently of the slicing direction and subsampling rate, the worst case performance is $O(\frac{t}{\sqrt[3]{b^2}})$. This implies that the fine resolution performance of the scheme is equivalent to the standard blocking scheme while at coarse resolutions it can get orders of magnitude better. More importantly, this allows one to produce coarse resolution outputs at interactive rates independent of the total size of the dataset.

A more accurate analysis can be performed to take into account the constant factors that are hidden in the big O notation and determine exactly which approach requires one to load into memory more data from disk. We can focus our attention to the case of a 8GB dataset with disk pages on the order of 32KB each as shown in diagram of Figure 1.5. One slice of data is 4MB in size. In the brick decomposition case, one would use $32 \times 32 \times 32$ blocks of 32KB. The data loaded from disk for a slice is 32 times larger than the output, that is 128MB bytes. As the subsampling increases up to a value of 32 (one sample out of 32) the amount of data loaded does not decrease because each $32 \times 32 \times 32$ brick needs to be loaded completely. At lower subsampling rates, the data overhead remains the same: the data loaded is 32768 times larger than the data needed. In the binary tree with Z-order remapping the data layout is equivalent to a KD-tee constructing the same subdivision of an octree. This maps on the slice to a KD-tree with the same decomposition as a quadtree. The data loaded is grouped in blocks along the hierarchy that gives an overhead factor in number of blocks of $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \cdots < 2$ while each block is 16KB. This means that the total amount of data loaded at fine resolution is the same. If the block size must be equal to 32KB the data located would twice as much as the previous scheme. The advantage is that each time the subsampling rate is doubled the amount of data loaded from external memory is reduced by a factor of four.

1.5.2 Tests with Memory Mapped Files

A series of basic tests were performed to verify the performance of the approach using a general purpose paging system. The out-of-core component of the scheme was implemented

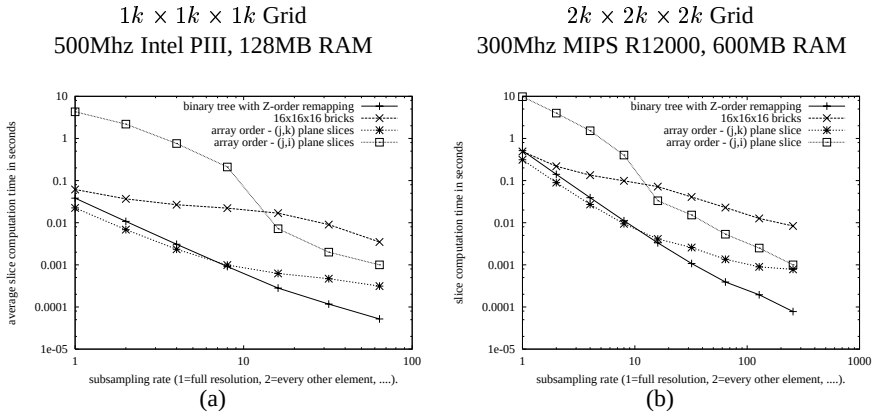


Fig. 1.6. Two comparisons of slice computation times of four different data layout schemes. The horizontal axis is the level of subsampling of the slicing scheme (test at the finest resolution are on the left).

simply by mapping a 1D array of data to a file on disk using the `mmap` function. In this way the I/O layer is implemented by the operating system virtual memory subsystem, paging in and out a portion of the data array as needed. No multi-threaded component is used to avoid blocking the application while retrieving the data. The blocks of data defined by the system are typically 4KB. Figure 1.6(a) shows performance tests executed on a Pentium III Laptop. The proposed scheme shows the best scalability in performance. The brick decomposition scheme with 16^3 chunks of regular grids shows the next best compromise in performance. The (i, j, k) row major storage scheme has the worst performance compromise because of its dependency on the slicing direction: best for (j, k) plane slices and worst for (j, i) plane slices. Figure 1.6(b) shows the performance results for a test on a larger, 8GB dataset, run on an SGI Octane. The results are similar.

1.6 Budgeted I/O and Compressed Storage

A portable implementation of the indexing scheme based on standard operating system I/O primitives was developed for Unix and Windows. This implementation avoids several application level usability issues associated with the use of `mmap`. The implemented memory hierarchy consists of a fixed size block cache in memory and a compressed disk format with associated meta-data. This allows for a fixed size runtime memory footprint, required by applications.

The input to this system is a set of sample points, arbitrarily located in space (in our tests, these were laid out as a planar grid) and their associated level in the index hierarchy. Points are converted into a virtual block number and a local index using the hierarchical Z order space filling curve. The block number is queried in the cache. If the block is in the cache, the sample for the point is 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 exceeds a user defined processing time limit. The cache is filled asynchronously by I/O threads which read

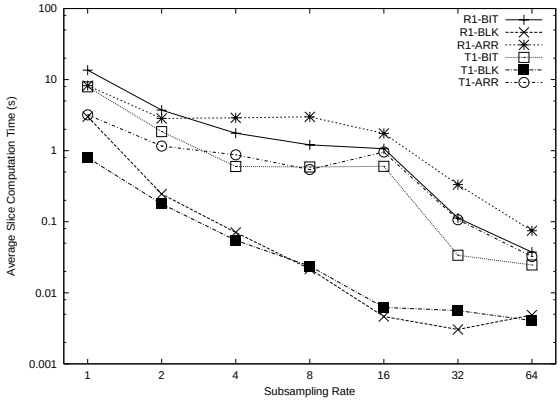


Fig. 1.7. Average slice computation time for 512^2 sample slices on a Linux laptop with a Intel Pentium III at 500Mhz using a 20MB fixed data cache. Results are given for two different plane access patterns `R1` and `T2` as well as three different data layouts `BIT`, `BLK` and `ARR`. The input data grid was $2048 \times 2048 \times 1920$.

compressed blocks from disk, decompress them into the cache, and resolve any sample points pending on that cache operation.

The implementation was testing using the same set of indexing schemes noted in the previous section: (`BIT`) our hierarchical space filling curve, (`BLK`) decomposition of the data in cubes of size equal to the disk pages and (`ARR`) storage of the data as a standard row major array. The dataset was one 8GB ($2048 \times 2048 \times 1920$) time-step of the PPM dataset [16] shown in Figure 1.11 (same results were achieved with the visible female dataset shown in Figure 1.12). Since the dataset was not a power of two so it was conceptually embedded in a 2048^3 grid and reordered. The resulting 1D array was blocked into 64KB segments and compressed using zlib. Entirely empty blocks resulting from the conceptual embedding were not stored as they would never be accessed. The re-ordered, compressed data was around 6 percent of the original dataset size, including the extra padding.

Two different slicing patterns were considered. Test `R1` is a set of one degree rotations over each primary axis. Test `T1` is a set of translations of the slice plane parallel to each primary axis, stepping throught every slice sequentially. Slices were sampled at various levels of sampling resolution.

In the baseline test on a basic PC platform, shown in Figure 1.7, with a very limited cache allocation, the proposed indexing scheme was clearly superior (by orders of magnitude), particularly as the sampling factor was increased. Our scheme allows one to maintain real-time interaction rates for large datasets using very modest resources (20MB).

We repeated the same test on an SGI Onyx2 system with higher performance disk arrays, the results are shown in Figure 1.8. The results are essentially equivalent, with slightly better performance being achived at extreme sampling levels on the SGI. Thus, the even the hardware requirements for the algorithm are very conservative.

To test the scalability of the algorithm, we ran tests with increased output slice size and input volume sizes. When the number of slice samples was increased by a factor of four (Figure 1.9) we note that our `BIT` scheme is the only one that scales running times linearly with the size of the output for subsampling rates of two or higher.

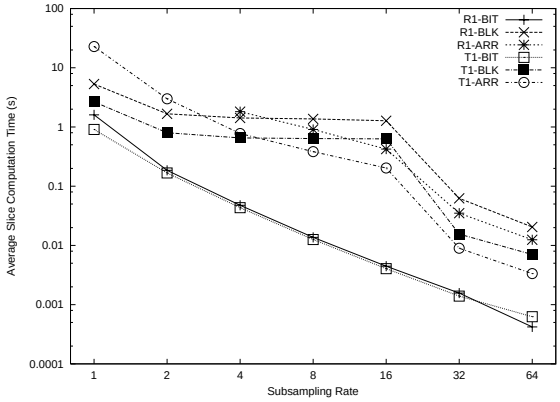


Fig. 1.8. Average slice computation time for 512^2 sample slices on an SGI Onyx2 with 300Mhz MIPS R12000 CPUs using a 20MB fixed data cache. Results are given for two different plane access patterns `R1` and `T2` as well as three different data layouts `BIT`, `BLK` and `ARR`. The input data grid was $2048 \times 2048 \times 1920$.

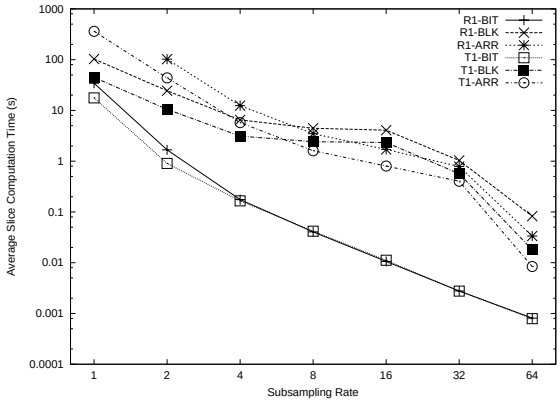


Fig. 1.9. Average slice computation time for 1024^2 sample slices on an SGI Onyx2 with 300Mhz MIPS R12000 CPUs using a 20MB fixed data cache. Results are given for two different plane access patterns `R1` and `T2` as well as three different data layouts `BIT`, `BLK` and `ARR`. The input data grid was $2048 \times 2048 \times 1920$.

Finally, a 0.5TB dataset ($8192 \times 8192 \times 7680$ grid) formed by replicating the 8GB timestep 64 times was run on the same SGI Onyx2 system using a larger 60MB memory cache. Results are shown in Figure 1.10. Interactive rates are certainly achievable using our indexing scheme on datasets of this extreme size and are very comparable to those obtained for a the $2k^3$ grid case (Figure 1.8) with this scheme.

Overall, results generally parallel those illustrated in the `mmap` experiments. Major differences stem from the increases in access time caused by the use of computationally expensive compression schemes and the potential for cache thrashing caused by the selection of (rela-

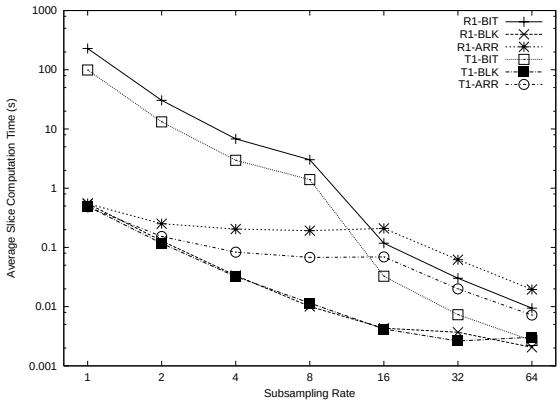


Fig. 1.10. Average slice computation time for 512^2 sample slices on an SGI Onyx2 with 300Mhz MIPS R12000 CPUs using a 60MB fixed data cache. Results are given for two different plane access patterns `R1` and `T2` as well as three different data layouts `BIT`, `BLK` and `ARR`. The input data grid was $8192 \times 8192 \times 7680$.

tively) small local cache sizes, particularly with schemes lacking the degree of data locality provided by our scheme.

1.7 Conclusions and Future Directions

This paper introduces a new indexing and data layout scheme that is useful for out-of-core hierarchical traversal of large datasets. Practical tests and theoretical analysis for the case of multi-resolution slicing of rectilinear grids illustrate the performance improvements that can be achieved with this approach, particularly within the context of a progressive computational framework. For example we can translate and rotate planar slices of an 8k cubed grid achieving half-second 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.

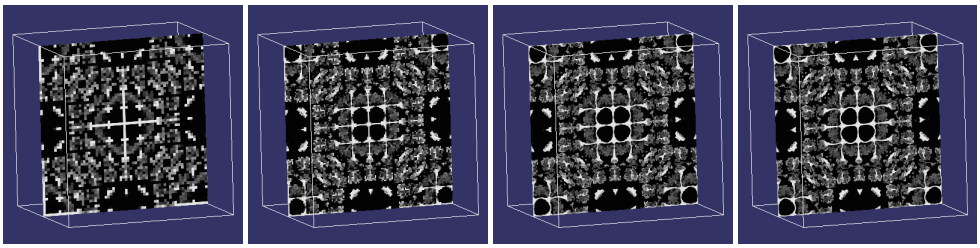
Future directions being considered include integration with wavelet compression schemes, the extension to general rectangular grids, distributed memory implementations and application to non-rectilinear hierarchies.

1.8 Acknowledgments

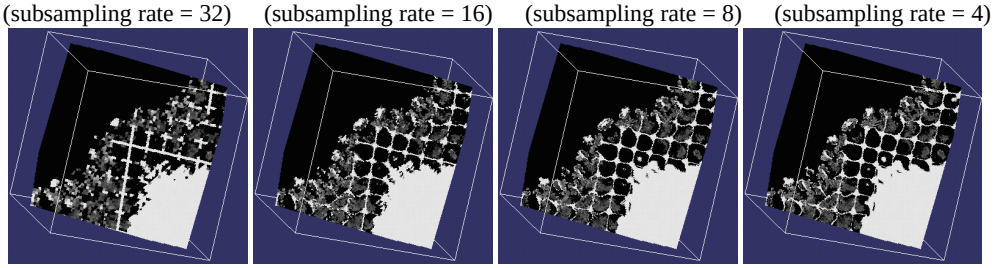
This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

References

1. James Abello and Jeffrey Scott Vitter, editors. *External Memory Algorithms and Visualization*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

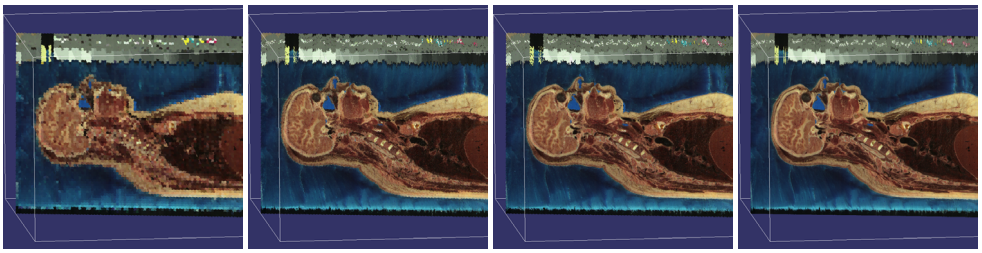


(subsampling rate = 32) (subsampling rate = 16) (subsampling rate = 8) (subsampling rate = 4)



(subsampling rate = 32) (subsampling rate = 16) (subsampling rate = 8) (subsampling rate = 4)

Fig. 1.11. Progressive refinement of two slices of the PPM dataset. (top row) Slice orthogonal to the x axis. (bottom row) Slice at an arbitrary orientation.



(subsampling rate = 64) (subsampling rate = 16) (subsampling rate = 4) (subsampling rate = 1)

Fig. 1.12. Progressive refinement of one slice of the visible human dataset. Note how the inclination of the slice allows to show at the same time the nose and the eye. This view cannot be obtained using only orthogonal slice.

2. Lars Arge and Peter Bro Miltersen. On showing lower bounds for external-memory computational geometry problems. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.
3. T. Asano, D. Ranjan, T. Roos, and E. Welzl. Space filling curves and their use in the design of geometric data structures. *Lecture Notes in Computer Science*, 911:36–44, 1995.
4. C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In Stephan N. Spencer, 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. L. Balmelli, J. Kovačević, and M. Vetterli. Quadtree for embedded surface visualization: Constraints and efficient data structures. In *IEEE International Conference on Image Processing (ICIP)*, pages 487–491, Kobe Japan, October 1999.
6. Y. Bandou and S.-I. Kamata. An address generator for a 3-dimensional pseudo-hilbert scan in a cuboid region. In *International Conference on Image Processing, ICIP99*, volume I, 1999.
7. Y. Bandou and S.-I. Kamata. 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.
8. Yi-Jen Chiang and Cláudio T. Silva. I/O optimal isosurface extraction. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization ́97*, pages 293–300. IEEE, November 1997.
9. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS '93)*, Palo Alto, CA, November 1993.
10. M. Griebel and G. W. Zumbusch. Parallel multigrid in an adaptive pde solver based on hashing and space-filling curves. 25:827:843, 1999.
11. D. Hilbert. Über die stetige abbildung einer linie auf ein flachenstück. *Mathematische Annalen*, 38:459–460, 1891.
12. S.-I. Kamata and Y. Bandou. An address generator of a pseudo-hilbert scan in a rectangle region. In *International Conference on Image Processing, ICIP97*, volume I, pages 707–714, 1997.
13. J. K. Lawder. *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.
14. J. K. Lawder and P. J. H. King. 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 Science*, pages 20–35. Springer Verlag, July 2000.
15. Y. Matias, E. Segal, and J. S. Vitter. Efficient bundle sorting. In *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA '00)*, January 2000.
16. A. Mirin. Performance of large-scale scientific applications on the ibm asci blue-pacific system. In *Ninth SIAM Conf. of Parallel Processing for Scientific Computing*, Philadelphia, Mar 1999. SIAM. CD-ROM.
17. B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of hilbert spacefilling curve. *IEEE Transactions on knowledge and data engeneering*, 13(1):124–141, 2001.
18. R. Niedermeier, K. Reinhardt, and P. Sanders. Towards optimal locality in meshindexings, 1997.
19. Rolf Niedermeier and Peter Sanders. On the manhattan-distance between points on space-filling mesh-indexings. Technical Report iratr-1996-18, Universität Karlsruhe, Informatik für Ingenieure und Naturwissenschaftler, 1996.
20. M. Parashar, J.C. Browne, C. Edwards, and K. Klimkowski. A common data management infrastructure for adaptive algorithms for pde solutions. In *SuperComputing 97*, 1997.
21. Hans Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.
22. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, March 2000.