A FAST MULTITHREADED OUT–OF–CORE VISUALIZATION TECHNIQUE*

Peter D. Sulatycke** and Kanad Ghose**
Dept. of Computer Science
State University of New York, Binghamton, NY 13902–6000
email: {sulat, ghose}@cs.binghamton.edu

Abstract: Out–of–core rendering techniques are necessary for viewing large volume disk–resident data sets produced by many scientific applications or high resolution imaging systems. Traditional visualizers can provide real–time performance but require all of the data to be viewed to be in the RAM. We describe a multithreaded implementation of an out–of–core isosurface renderer that does not impose such restrictions and yet provides performance that scales well with the size of the data. Our renderer uses an interval tree data structure on disk with a layout that reduces disk seeks to read out only the relevant data from the disk. The low resulting disk latencies are hidden by using prefetching and multithreading to overlap the activities of the rendering computations and disk accesses. Our renderer outperforms the out–of–core isosurface renderer of the well–known vtk toolkit by about one order of magnitude and several orders of magnitude when compared against vtk toolkit's optimized in–core algorithm on large representative CT scan data. The multithreaded version also scales well with the number of threads.

Keywords: multithreading, out–of–core visualization, isosurface rendering.

1. INTRODUCTION

Modern scientific applications generate very large three–dimensional data sets, also commonly referred to as volume data. Volume data can be generated directly from imaging systems (such as CT, MRI, PET and Ultrasound scans) or it can be "synthesized" by computer simulations (such as in Computational Fluid Dynamics (CFD) and Pharmacology). Whatever the origin, the resulting size of the data makes it necessary that the data be visualized to be fully understood. Several techniques are used to visualize volume data in modern visualization system, the most common being isosurface extraction. Isosurface extraction can be simply described as the process of producing a three–dimensional surface with a constant value from a volume data set.

Generally, for isosurface extraction, the volume data is considered as a collection of polyhedra or cells, sharing adjacent data points within a 3–dimensional lattice. Isosurface extraction typically involves three stages (1) the identification of cells that intersect the desired isosurface (i.e., cells that contain points where the isovalue occurs), (2) determining and modeling (typically with triangles) the exact manner in which the isosurface intersects these cells and (3) the actual rendering of the three–dimensional isosurface as a two–dimensional image on a CRT screen.

The last stage of isosurface extraction is usually performed by some type of graphics accelerator hardware, thus this paper is only concerned with stages 1 and 2. The second stage is typically implemented as the well established marching cubes algorithm [LoCl 87]. This algorithm uses a lookup table to determine the triangles that form the isosurface within an individual cell. This technique is successful on small data sets but must be parallelized to be effective on large data sets [HaHi 92, Ma 92]. A multithreaded version of the marching cubes lookup is used in the approach presented in this paper.

The thrust of recent research has involved stage 1, the locating of cells intersected by the isosurface [Ga 91, ItKo 94, LSJ 96, ShJo 95, SHLJ 96, CMM+ 97]. The algorithm described in [CMM+ 97] is particularly effective because it obviates the need for cell searching, only cells that actually contribute to the isosurface are visited. This sought after property is realized through the use of an interval tree data structure [Ed 80]. Stage 2 of the rendering generally uses the marching cube algorithm or its variations. These isosurface extraction algorithms generally assume the availability of enough fast local memory to store the complete data set and its auxiliary structures. To visualize any data set larger than the available RAM capacity, the user has to add more RAM or rely on virtual memory mechanisms to move data through the storage hierarchy. Specifically, when conventional visualization techniques are applied to visualize disk–resident volume data whose capacity exceeds that of the RAM, severe page thrashing occurs, causing the visualization time to increase unduly and well beyond real–time bounds. Consequently, the performance of these renderers do not scale well with the data size.

The visualization of large data sets that exceed the RAM capacity by a significant amount have led researchers to devise out–of–core visualization algorithms whose performance are not limited by the available RAM capacity. Such out–of–core algorithms are also essential for viewing very large data sets (typically a few hundred megabytes to several gigabytes or more in size).

Proposed out–of–core visualization techniques have relied on several techniques to improve performance. Unique data structures on disk (k–ary interval trees [ChSi 97a], metablock trees [ChSi 97b]) that reduce the number of disk I/O operations have been used in this respect. Also used are spatial data structures on disk that exploit locality [USM 97]. Application controlled demand paging [CoEl 97] has

* patent pending on several aspects of this work
** also affiliated with 3DVis Technologies, Inc.

also been used to explicitly control data movement between the disk and the RAM.

Out–of–core techniques avoid performance degradation due to thrashing, but to achieve reasonable performance out–of–core visualization techniques must attempt to reduce the disk access time. Existing techniques focus on reducing the number of disk I/O operations; they fail to additionally optimize the number of disk seek operations. This is an important consideration for modern disks whose improvements in data access rates have not been accompanied by a commensurate decrease in disk seek times.

This paper describes a multithreaded technique for visualizing large disk–resident volume data sets in real–time or near real–time, overcoming the performance limitations of in–core visualizers as well as performing better than existing out–of–core visualizers. Our out–of–core renderer achieves this performance by using multithreading to overlap I/O and rendering computations and by using disk data structures that optimize the number of disk I/O operations, the amount of data involved in disk I/O, and more importantly, the number of disk seeks. In the sequential implementation of our technique described in [SuGh 98], disk I/O is no longer the bottleneck; rendering computations actually limit the performance. This paper focuses on the use of multithreading to hide disk I/O latencies and the reduction of the computational overhead of our sequential implementation. In fact, even with multiple threads processing data that has been prefetched, the disk latency of the I/O thread is completely hidden. The performance of our multithreaded implementation on a small scale SMP scales well with the number of processors and thus further improves the performance of our sequential out–of–core visualizer. This implementation not only achieves real–time or near real–time performance of isosurface rendering of very large, disk–resident volume data but also does it with a minimum of memory. This frees up the RAM for use by other aspects of the visualization process.

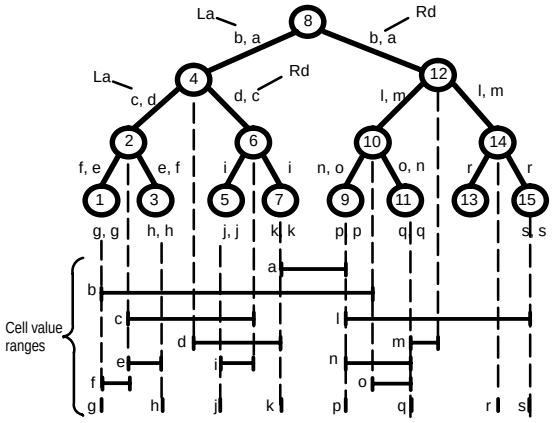
2. BINARY INTERVAL TREES ON DISKS

We make use of the binary interval tree data structure on disk, with an layout that optimizes disk seeks to hold the volume data. We first provide a brief overview of the in–core binary interval, as used in isosurface extraction. The reader is referred to [Ed 80, CMM+97] for further details. The interval tree is a data structure that has been shown to be optimal for the so–called stabbing point querying problem. The stabbing point query problem directly corresponds to finding all cells that contain an isovalue. The problem of extracting an isosurface for an isovalue Q can be generically formulated as the following problem:

2.2 Organizing Interval Trees on Disk for Fast Accesses

The creation of our out–of–core interval tree begins with the standard binary interval tree. An example of a 16–node binary interval tree in linked list form is given in Figure 1 (a). We directly produce a flattened, pointerless version of this tree, without compromising the interval tree's optimal searching capabilities. We store the length of every node, even if its zero, in a separate file from the interval tree file. The interval tree construction is a one–time "pre–processing" step for our out–of–core visualization technique; once constructed, it can be used for all subsequent visualization of the data.

The data layout of each subtree of the interval tree is unique: the right subtree uses a preorder layout (root, left tree, right tree) while the left subtree uses a variation of this (root, right tree, left tree). Figure 1 (b) depicts the layout of the resulting structure on the disk. This layout allows the out–of–core data to be accessed in order, without the need for extra seeks. The data lists within a node, including all cell information (density, gradient, location), are written in La, Rd order. To avoid having to load in all of a list's data, cell data is duplicated for both the La and Rd list. All list data is then written sequentially to disk in binary format; no padding, pointers, or secondary file structures are used.



(a) Binary interval tree of cell value ranges

La, Rd* for each node	ba	cd, dc		k	j	fe, ef	h	g	ba	lm, lm	no, on	p	q	r	–	s
Node #	8	4	6	7	5	2	3	1	8	12	10	9	11	14	13	15
Length of La or Rd	2	2	1	1	1	2	1	1	2	2	2	1	1	1	0	1

* For leaf nodes, La =Rd, so only one of these lists are stored

(b) Flattened storage representation

Figure 1. The binary interval tree and its storage format for in–order disk accesses

Tree traversal is easily accomplished by preloading the length file into RAM. This file is very small since the number of nodes is limited; 256 in our case. Keeping lengths separate from the interval tree facilitates tree traversal and prefetching. Unlike the k–ary interval tree of Chiang and Silva [ChSi 97a] we do not require any kind of searching in a node to determine the address of the next node to visit. We just add up node lengths and then move directly to the node of interest.

3. MULTITHREADED IMPLEMENTATION

We make use of a single I/O thread and several computation threads in our implementation. These threads interact through a set of buffers, with the I/O thread acting as a producer and the computation threads as the consumers. In the sequential implementation of our out–of–core visualizer the computation process turns out to be more

time consuming than the disk I/O process; this is confirmed by the measurements from our sequential implementation [SuGh 98], as described later. It is thus necessary to use multiple computation threads to achieve a balance of activity with a single I/O thread. The producer thread (i.e., the I/O thread) and the consumer threads overlap to effectively hide disk latency and prevent the CPUs from stalling for lack of data, as seen from the results presented.

3.1 The I/O Thread

The I/O thread is responsible for traversing the interval tree on the disk and prefetching disk data blocks that contain cells that match the query isovalue into the buffers. To maintain efficient disk access, all disk reads equal the low level file block size. Thus the size of the buffers are multiples of the file block size. In addition, file seeking is kept to a minimum by seeking directly to a node and not for file navigation purposes. Some of the cells loaded into the buffers may not be intersected by the surface of interest due to the exclusive use of block reads. This is caused by the fact that block reads do not always fall on cell boundaries. Due to this, the file I/O thread does a very simple preprocessing of the data by marking the data within the buffer with flags and length indicators. Each flag indicates how the compute threads are to process *n* bytes of fetched data (*n* corresponding to the length indicator). The set of flags used for this purpose are: *skip*, *read_all*, *check_if_greater_than*, *check_if_less_then.* The majority of the computations are done within the compute thread, allowing the I/O thread to initiate I/O requests as quickly as possible.

On reading in a file block of data, the file I/O thread checks to see if the last cell within the block meets the query criteria. (Many cells can fit within a disk block.) Leading parts of the disk block data may be skipped, since list boundaries do not coincide with disk block boundaries. If the last cell in the block meets the search criterion, a flag and length indicator are set indicating that all of the cells within the block meet the query criteria and should be rendered by the compute threads.

If the last cell is invalid then the flag is set indicating that all cells within the block must be checked by the compute thread that will process the buffer containing this data. Note this is different from querying an in–core interval tree where cell lists are strictly checked in ascending or descending list order. The non–padded nature of the file causes block reads to possibly start before the beginning of a node list and possibly end in the middle of a cell. This requires the file I/O thread to set a flag indicating the number of bytes to skip to the beginning cell. It also requires some minimal overhead to keep track of partial cells.

In the worst case the file I/O thread only executes 2*TREE_HEIGHT (16 in our case) more read operations than what would be needed if the query matching data were completely in order. This improves upon the optimal number of I/O operations reported by Chiang and Silva in [ChSi 97a], where additional pointers must be read and extra seeks performed. While the I/O thread is reading data into an empty buffer, the compute threads check flags and process data in the buffers previously filled by the I/O thread. The compute threads operates directly on the data within the buffer, no copying or moving of data is performed. This improves performance by avoiding cache misses, page remapping and unnecessary loads and stores.

3.2 The Computation Threads

The computation threads are responsible for producing a polygonal representation of the surface passing through each of the valid cells in the buffers. Each compute thread processes a buffer filled in by the I/O thread as follows. After checking a flag, the compute thread will either skip *n* bytes, check in ascending/descending order cells that take up *n* bytes, or render all cells in the *n* bytes. The compute thread then goes onto the next flag repeating the process until all data within the buffer is exhausted. The compute thread then proceeds on to process the next filled buffer; if none exists at that point, its waits for the file I/O thread to fill in a buffer or for the termination of processing, whichever occurs earlier.

A compute thread renders a cell by using the marching cube algorithm [LoCl 87] to calculate the triangles and corresponding normals that make up the surface within the cell. The triangles are then copied directly into the buffer of a graphics engine where shading and displaying is performed.

3.3 Synchronization of Producer and Consumers

The single producer and multiple consumers share a circular array of N buffers. While the producer thread is prefetching data from disk into one of these buffers, the consumer threads are processing data that has already been read into some of the other buffers. All I/O and processing is directly performed on these buffers, no copying of data is done. To synchronize the interaction of these threads the following mutex protected variables are used:

in:	global buffer index, pointing to the next buffer available to the producing thread
out:	global buffer index, pointing to the next buffer available to a consuming thread
buf_done:	flag for each buffer, initially set to TRUE
num_empty:	global buffer counter initially set to the number of buffers
num_full:	global buffer counter initially set to zero

The producing thread fills the N buffers in a circular fashion, using the variable “in” to indicate which buffer to write to. The consuming threads vie for the next available buffer, indicated by the “out” variable. Thus the “in” and “out” variables delineate the tail and head of active buffers within the N buffers. Since the consuming threads can complete in any order, a “buf_done” flag for each buffer must be used to indicate consuming thread completion. Without this flag the producing thread may try to write to the same buffer that is still being consumed. The counters “num_empty” and “num_full” are used to indicate the number of full and empty buffers to the producing and consuming threads respectively.

When a consuming thread wants to acquire a buffer it checks to see if “num_full” is non–zero. If “num_full” is zero then the consuming thread waits on a conditional wait, which in time will be awoken when “num_full” is non–zero. If “num_full” is non–zero the consuming thread will process data in the buffer pointed to by “out”. When

the producer wants to write to a new buffer, it checks the global "num_empty" counter to see if any of the buffers are empty.

If "num_empty" is zero the producing thread waits on a conditional wait, which in time will be awakened when "num_empty" is non–zero. If "num_empty" is non–zero then the producer goes to the buffer pointed to by "in" and checks to see if its "buf_done" flag is marked as TRUE. If this buffer's "buf_done" flag is set to FALSE then the producer waits on a conditional wait, which in time will be awoken when this "buf_done" flag is TRUE. If the "buf_done" flag is TRUE the producer will write data into this buffer. The producer must check both "num_empty" and the associated "buf_done" flag before it can do any writing of data. Without both layers of checks the producing thread may interfere with a consuming thread.

4. RESULTS

We implemented the multithreaded version of our out–of–core visualizer on Sun Solaris multiprocessor platforms – 4–CPU SPARC 20s and 2–CPU UltraSPARC 2s. The large volume data sets we used in our studies were the rectilinear CT scan data for the human head, "Head" (256 X 256 X 113 voxels) and an engine block, "Engine" (256 X 256 X 110 voxels), both obtained from the Univ. of North Carolina. The execution times presented here do not include any time for shading or displaying since they are done in parallel by the graphics card's rendering engine. All execution times were obtained as the harmonic means of several runs.

Before we present the results from the multithreaded implementation, it is instructive to look at the results of the sequential, non–threaded implementation. Figure 2 depicts the overall rendering time as a function of the number of cells on the isosurface for the Engine data set for our sequential implementation and a conventional isosurface renderer from the vtk toolkit (the optimized/patented version, labelled as VTK) [SML 97]. The same figure also compares our sequential implementation against an out–of–core isosurface renderer (VTK–I/O), also from the vtk toolkit, which reads in the cell data slice by slice from the disk. In the runs we made to generate the results for Figure 2, the CPU and disk caches were initially flushed before the first isosurface extraction. The implementations are on a SPARC 20 multiprocessor workstation with 32 MBytes of dRAM, whose multiprocessing capabilities are not obviously exploited in these implementations. The performance advantages of our sequential version are quite obvious – as the data set size for the isosurface increases beyond the available RAM capacity, the rendering times for both VTK and VTK–I/O go up significantly. The situation is particularly worse for VTK, which induces a significant amount of page swapping. The main observation from Figure 2 is that our sequential technique performs at least an order of magnitude better than the well–known out–of–core renderer from the VTK–I/O toolkit. Similar results are observed for the other data sets.

To understand the bottlenecks in our sequential implementation, so that we can speed it up further using multithreading, its useful to look at Figure 3, which depicts the disk I/O times in VTK–I/O and our technique as a function of the data set size. The absolute values of I/O times in our technique are higher than VTK–I/O's since the interval tree data structure has a higher data size. Note also that as VTK–I/O reads in the volume data from the disk slice by slice irrespective of the isovalue, it can exploit disk caching well, as seen from the drop in its disk I/O time when the number of cells on the isosurface from 107560 to 196224. In contrast, our technique locates the appropriate data set on the disk by traversing the interval tree structure, as described earlier in Section 3, so it does not exploit disk caching as well as VTK–I/O. The I/O times in both techniques are, however, still low in absolute terms, but the I/O overhead is *relatively* higher for our technique. Also, the computational overhead in VTK–I/O is relatively very high – this is because the data read into the RAM from the disk has to be searched to locate the cells of relevance. In contrast, the need for searching in our technique is obviated by organizing the volume data as an interval tree on the disk in a manner that minimizes the disk seek time despite the much enhanced size of the interval tree structure. The discussions in this paragraph suggests what has to be done in the multithreaded implementation:

- Multithreading has to be used to hide disk I/O times. A single thread performing the disk I/O is sufficient, since disk access conflicts may be introduced with multiple threads performing disk I/O.
- Multiple "computation" threads will be used to not just overlap I/O activities with computation, but also to reduce the overall computational overhead in the sequential implementation.

In the subsequent sections, we describe the results pertaining to our multithreaded implementation. Unless qualified otherwise, the results are on a 4–processor SPARC 20 workstation with 32 MBytes of dRAM. Some results on a UltraSPARC 2 with 256 MBytes of dRAM are also included.

An effective manner of interaction among the threads is a necessity in our implementation. This essentially boils down to choosing the number of buffers and the size of each buffer. Recall that each compute thread works on the contents of a buffer independently. If the number of buffers are too small, the threads may run into each other and long waiting times may results for each thread. A small number of buffers also make it difficult to achieve good load balancing among the threads. Making the number of buffers too large can increase the thread synchronization overhead and make the I/O thread fall behind (whose performance is limited by the disk bandwidth). The size of each buffer is also critical. Making each buffer too small increases the synchronization overhead, since each thread will need to synchronize often after quickly filling up or consuming the buffer. Making each buffer too small also makes it difficult to amortize the flat startup overhead of a disk transaction. If the size of each buffer is too large, load balancing among the threads becomes difficult. It is thus likely that an optimum number of buffers, as well as an optimum buffer size exists. What we really want are the optimal number of buffers and the optimal buffer size that will be a good choice for a wide range of applications.

To determine an optimal size and number of buffers needed, we ran a large number of experiments to record the overall rendering time by varying the number of threads, number of processors, isovalues and the datasets. We determined that a range of buffer sizes, as well as a range of buffer numbers that are universally optimal. Some of these results are now presented. Figures 4 (a) and (b) show how the total rendering time as well as the time needed by the threads in our technique, vary with the size of each buffer, keeping the number of buffers and the number of compute threads fixed. In these figures, we show the minimum and maximum execution time, excluding any waiting times, of the compute threads. The time for the I/O thread also excludes its waiting times. (Note that, obviously, the total rendering time is not the sum of the individual thread times because of the overlapping of the activities of the threads in the multiprocessor system.) In between every isosurface extraction, the disk and processor caches were flushed to avoid ambiguities due to caching. The results in these figures are representative of all the other cases. A wide range of buffer sizes (75 to 500 disk blocks per buffer) seems to be optimal not just for the parameters shown but also for a wide range of the number of threads, data sets, as well as platforms. The various threads have close execution times – i.e., the threads are well–balanced for buffer sizes within this range.

Figures 5 (a) and 5 (b) show how the total rendering time as well as the thread times vary, with the number of buffers as the other parameters are kept fixed. These results are again representative of the other cases. As expected, when the number of buffers are small, the waiting times for the threads go up due to contention over the buffers, prolonging the overall rendering time. Again, 20 to 60 buffers, with a size of 150 disk blocks per buffer seems to work out well for a large number of combinations of number of threads, data sets, platforms and isovalues. As in the previous set of runs, the CPU and disk caches were flushed to avoid ambiguities due to caching. Note that we are not claiming that these numbers are optimal for each and every possible combination of data value, data sets, platforms or data characteristic – indeed, in some cases experiments similar to what we described above will be needed to determine a buffer size and number of buffers that work out best for the specific application. All the subsequent results use the optimal combination of 20 buffers, each with a capacity of 150 disk blocks. These are the numbers that have worked out to be a good combination for a wide range of variables that we have tested.

Figure 6 shows how performance of our multithreaded implementation varies with the number of compute threads on a 4–processor SPARC 20 and a 2–processor UltraSPARC system. As expected, when the number of compute threads are low, the I/O thread has to wait longer since each compute thread takes a longer time to complete the processing, thus slowing down the rate at which buffers are emptied. Put in other words, this simply implies that the computation bottleneck remains. With an increase in the number of compute threads, the I/O thread's execution time increases slowly due to contention over the shared variables. (The contention times are not part of the thread waiting time; the latter is the time spent by a thread waiting on a condition queue.) With a further increase in the number of computation threads, the thread scheduling overhead dominates, so neither the compute threads, nor the I/O thread can do as much useful processing, so the overall rendering time goes up. As seen from Figure 6, 4 compute threads and a single I/O thread (which uses the CPU relatively less compared to the compute thread) provide the best overall performance on the 4–CPU platform, while 2 to 3 compute threads provide the best performance for the 2–CPU platform. These figures also depict how well the performance scales with the number of compute threads when the number of compute threads increase from 1 to 4 on the SPARC 20. Similar trends are observed for the results on the UltraSPARC. The results for other isovalues and other data sets are also similar and not reported here for the sake of brevity. The general observation is that the use of prefetching and multithreading produces a scalable implementation and effective speedups, making it possible to achieve real–time performance with our out–of–core visualizer.

In Figure 7, we depict how the multithreaded implementations compare with the VTK and VTK–I/O codes for two isosurfaces whose cell count differ considerably. Significant performance gains are realized with the multithreaded implementation when the data set size is too big to be held in core, as seen for the SPARC 20 implementation (left graph) for the isosurface intersecting 659644 cells. At this cell count, VTK clearly induces serious page thrashing; VTK–I/O shows a similar behavior, although not as pronounced. The results for the UltraSPARC (right graph) clearly shows that even when the data fits into the RAM, our multithreaded implementation does significantly better than both VTK and VTK–I/O. Further, the absolute numbers for the rendering time with the multithreaded implementations on both platforms meet the real–time goals. Although not depicted here for the sake of brevity, the results for the Head data are quite similar.

Fundamental to our technique is a preprocessing step that organizes the volume data set into an interval tree. Once organized into an interval tree, all possible isosurfaces can be generated in any combination or order. Using a disk–to–disk technique, we can construct an interval tree for large data sets in time linear with the size of the data. For the data sets used in this experiment, the interval tree generation times are at most about 3 minutes.

5. CONCLUSIONS

We presented a multithreaded implementation of an out–of–core isosurface rendering system that we introduced in [SuGh 98]. When the data set size exceeds the available RAM capacity, our sequential implementation outperforms the out–of–core isosurface renderer of the well–known vtk toolkit (VTK–I/O) by about one order of magnitude and several orders of magnitude when compared against vtk toolkit's in–core algorithm (VTK). Additionally, when the data set size fits within the available RAM capacity, our out–of–core algorithm outperforms both the VTK–I/O and the VTK renderer by about one order of magnitude. The multithreaded version of our renderer does even better, scaling well with the

number of threads used, when appropriate number of processors are available. The total time needed by our rendering technique grows slowly with the size of the data set in contrast with VTK–I/O and VTK. The rendering times realized with our multithreaded renderer are a few seconds even with large data sizes and thus the performance can be considered as meeting real–time requirements. The advantages of our technique are as follows:

- Our out–of–core visualization technique lends itself to multithreading relatively easily. Significant performance gains are possible with the use of a few threads on SMPs.
- Unlike normal paging activity (where the application is swapped out and considerable time is spent in selecting a victim and adjusting the page mapping tables before a page I/O is initiated), our technique explicitly initiates disk I/O activity, keeping the application active.
- Our technique exploits concurrency across the I/O and compute threads. Furthermore, the prefetching used by the I/O thread is useful in hiding the disk access time.
- The in–order disk accesses inherent in our technique is also instrumental in cutting down the disk I/O time. In–order accesses not only reduce the disk I/O time but also increase the chances of I/O requests to be combined at the level of disk scheduling, amortizing the startup overhead.
- Our technique makes a very conservative use of the RAM capacity unlike most in–core or out–of–core visualizers. This effectively frees up RAM capacity for other uses.

We are currently investigating techniques for reducing the disk data size significantly and for exploiting the freed–up RAM capacity to further speed up our renderer.

References

[ChSi 97a] Y. Chiang, C. Silva, "I/O Optimal Isosurface Extraction", Proceeding of Visualization '97, pp 293–300, Phoenix AZ, Oct 1997

[ChSi 97b] Y. Chiang, C. Silva, "Isosurface Extraction in Large Scientific Visualizatino Applications Using the I/O–filter Technique", yet to be published.

[CMM+ 97] P. Cignoni, P. Marino, C. Montani, E. Puppo, R. Scopigno, "Speeding Up Isosurface Extraction Using Interval Trees", Visualization and Computer Graphics, vol. 3, no. 2, pp. 158–170, April 1997

[CoEl 97] M. Cox, D. Ellsworth, "Application–Controlled Demand Paging for Out–of–Core Visualization", Proceeding of Visualization '97, pp 235–244, Phoenix AZ, Oct 1997

[Ed 80] H. Edelsbrunner, "Dynamic Data Structures for Orthogonal Intersection Queries," Technical Report F59, Inst. Informationsverarb., Tech. univ. Graz, Graz, Austria, 1980.

[Ga 91] R.S. Gallagher, "Span Filter: An Optimization Scheme for Volume Visualization of Large Finite Element Models," Proc. Visualization '91 Conf. Proc., pp 68–75, 1991

[HaHi 92] Hansen, C.D. and Hinker, P., "Massively Parallel Isosurface Extraction," in Proceedings of Visualization '92, Kaufman, A.E. and Nielson, G., editors, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 77–83.

[ItKo 94] T. Itoh and K. Koyyamada, "Isosurface Generation by Using Extrema Graphs," Proc. Visualization '94. Los Alamitos, Calif.: IEEE Press,. 1994, pp. 77–83.

[Ma 92] P. Mackerras, "A Fast Parallel Marching–Cubes Implementation on the Fujitsu AP1000," Technical Report TR–CS–92–10, Department of Computer Science, Australian National University, August 1992.

[LSJ 96] Y. Livant, H. Shen, and C. Johnson, "A Near Optimal Isosurface Extraction Algorithm for Structured and Unstructured Grids," IEEE Trans. Visualization and Computer Graphics, vol.2, no. 1, pp. 73–84, Apr. 1996

[LoCl] 87] W.E. Lorensen and H.E. Cline, .arching Cubes: A High Resolution 3D Surface Construction Algorithm, Computer Graphics, vol. 21, no. 4, pp.163–169, July 1987

[SML 97] Schroder, W., Martin, K. and Lorensen, B., *The Visualization Toolkit: An Object–Oriented Approach to 3–D Graphics*, 2nd Edition, Prentice–Hall, 1997.

[ShJo 95] H. Shen and C.R. Johnson, "Sweeping Simplicies: A Fast Iso–Surface Extraction Algorithm for Unstructured Grids," Proc. Visualization '95, 1995.

[SHLJ 96] H.Shen, C.D.Hansen, Y. Livnat, and C.R. Johnson, "Isosurfacing in Span Space with utmost Efficieny (ISSUE)" Visualization '96 Conf. proc., pp. 287–294, Oct. 1996.

[SuGh 98] P. Sulatycke and K. Ghose, "Out–of–Core Interval Trees for Fast Isosurface Extraction", in Proc. Late Breaking Topics, IEEE VIsualization '98 Conference, pp. 25–28. (Available at: http://opal.cs.binghamton.edu/~sulat.)

[USM 97] S. Ueng, C. Sikorski, K. Ma, "Out–of–Core Streamline Visualization on Large Unstructured Meshes", IEEE Trans. Visualization and Computer Graphics, vol.3, no. 4, pp 370–380, 1997.

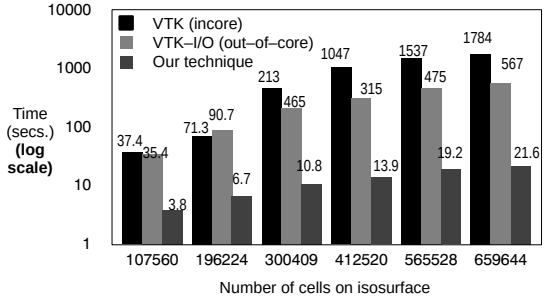


Figure 2. Rendering time as a function of the number of cells on the isosurface in the **sequential** implementation. The data rendered is "Engine".

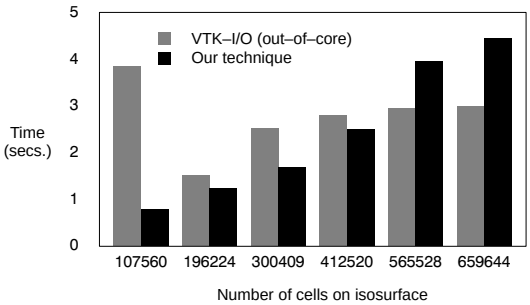


Figure 3. Disk I/O time as a function of the number of cells on the isosurface in the **sequential** implementation. The data rendered is "Engine".

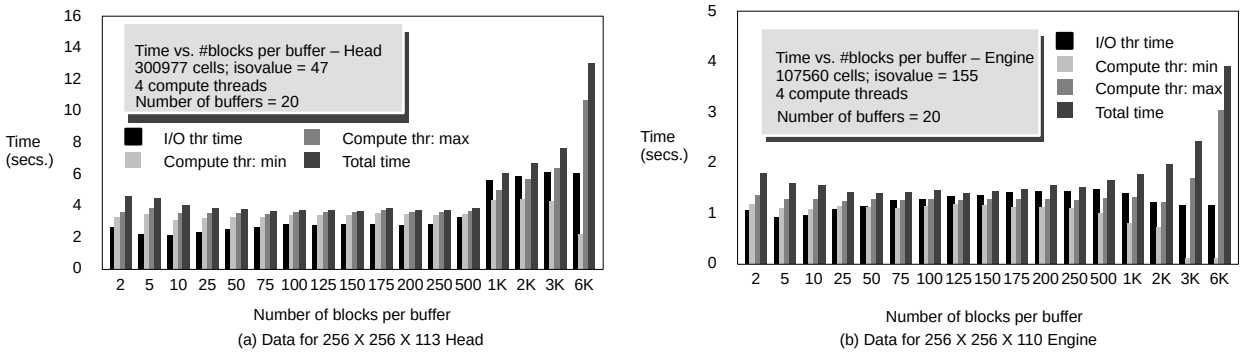


Figure 4. Total rendering time and thread execution times as a function of the number of disk blocks per buffer.

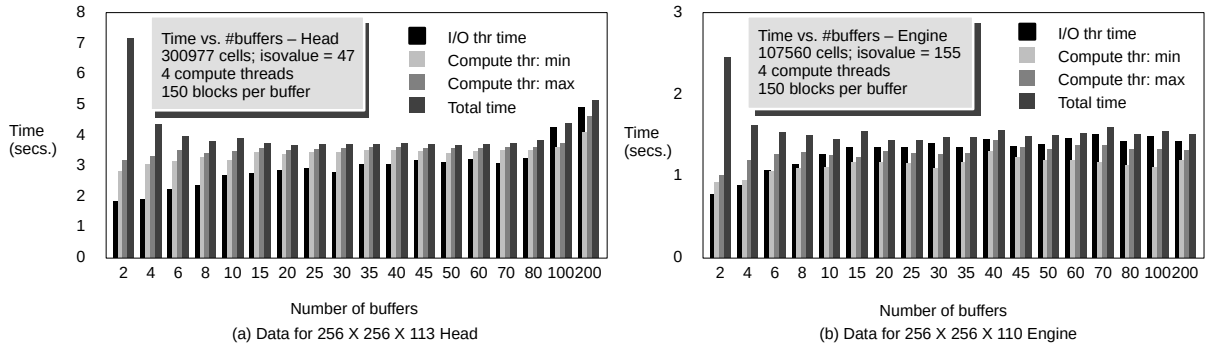


Figure 5. Total rendering time and thread execution times as a function of the number of buffers.

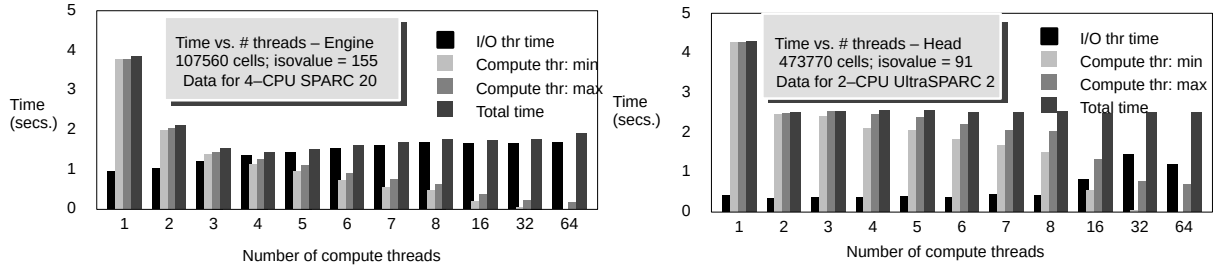


Figure 6. Total rendering time and thread execution times as a function of the number of threads.

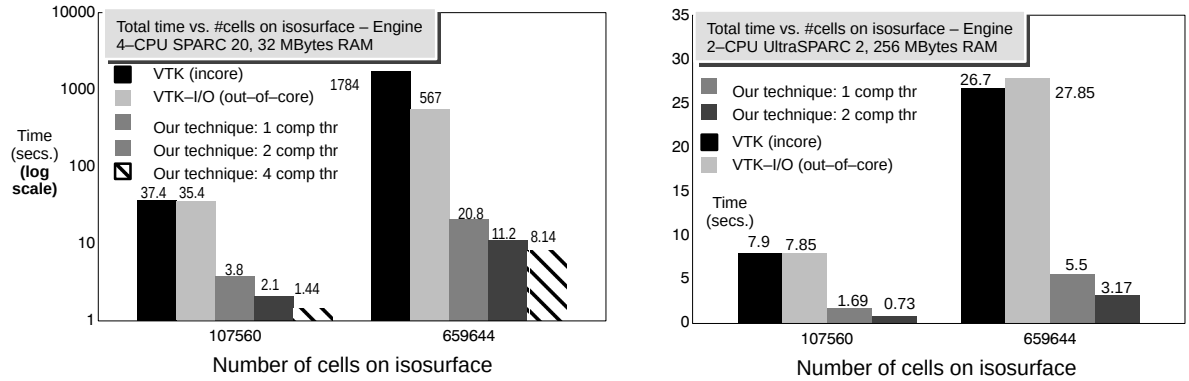


Figure 7. Rendering time as a function of the number of cells on the isosurface in various implementations. The data rendered is “Engine”.