Eurographics Symposium on Parallel Graphics and Visualization (2004)
Dirk Bartz, Bruno Raffin and Han-Wei Shen (Editors)

Fast Remote Isosurface Visualization With Chessboarding

A. Neeman, P. Sulatycke, and K. Ghose

State University of New York, Binghamton

Abstract

Isosurface rendering is a technique for viewing and understanding many large data sets from both science and engineering. With the advent of multi-gigabit-per-second network backbones such as Internet2 and fast local networking technology, scientists are looking at new ways to share and explore large data sets remotely. Telemedicine, which encompasses both videoconferencing and remote visualization is likely to be in widespread use with these advances. Despite the availability of increased bandwidth, two challenges remain. First, the time it takes to locate cells intersecting an isosurface of interest must be reduced for large data sets; a cell extraction technique that scales with data size is also critical. The second challenge has to do with the mitigating the effects of network latency on the overall isosurface visualization time. We present a remote isosurface visualization technique that addresses these two challenges. Isosurface extraction delays are reduced through the use of a search-optimized, chessboarded interval tree data structure on the disk. Network transport delays are reduced by sending cells extracted from the chessboarded data on the server, compressing it by about 87%. In addition, network transport delays are hidden effectively by overlapping data transport with server side functions. On a 100 Mbits/sec. switched LAN, the remote visualization time - the time between the issue of a query from the client side to the server and the displaying of a complete image on the client is only a few seconds for most isovalues in the well-known visible woman data set.

1. Introduction

Isosurface visualization is commonly used for CT (Computer Axial Tomography) and MRI (magnetic resonance imaging) data in medicine as well as CFD (Computational Fluid Dynamics) in many engineering disciplines. Isosurface visualization is one of several techniques to retrieve data of interest and change it to a graphical form.

This paper deals with remote isosurface visualization: viewing an isosurface for data stored on a server and sent to a remote client. Two challenges are encountered in the process of remote isosurface visualization. The first challenge is one encountered by all isosurface visualization processes that have to deal with large data sets. As the raw data set increases in size and exceeds the available RAM capacity, severe page thrashing can seriously degrade the performance of cell extraction, the phase that locates cells that intersect the isosurface of interest. Often, to speed up the cell extraction phase, the raw data set is transformed into search-optimized structures that are usually higher in size compared to the raw data set, which exacerbates the thrashing problem. The second challenge in remote visualization has to do with the network latency. Despite the increasing network bandwidth, challenges remain to making remote visualization fast and simple. Bottlenecks occur at the fringes of the network, affecting end to end performance. Distance induced latency also causes performance problems for data intensive applications. Finally it should be noted that the network is a shared medium, and there are no quality of service guarantees at this time.

We present a technique that addresses the two challenges for remote isosurface visualization. We use a search optimized interval tree data structure [Ede80] for cell extraction, as first reported in [CMM*97]. However, unlike the approach taken in [CMM*97], the interval tree data structure is stored on the disk in a format that reduces disk seeking time drastically. The resulting cell extraction performance is very close to what one would obtain if the raw or transformed data set could all fit into the RAM [SG98],[SG99]. This is achieved by successfully hiding the disk latency using multiple threads and the seek-reduced layout. To address the network latency, we adapt the chessboarding scheme of [CMM*97] to remove redundant information about the

cells. The data for the corner of a cell is shared with 7 other cells. Chessboarding removes this redundancy. However, the chessboarding technique of [CMM*97] cannot be used for the out-of-core interval tree, as one cannot randomly access cell data on a disk. Consequently, our chessboarding technique, although functionally similar to that of [CMM*97], is virtually a new approach. We reduce the impact of network delays into ways: by performing cell extraction of chessboarded data on the server and sending the compressed data to the client and by overlapping cell extraction with network transport delays. In effect, chessboarding is a lossless compression technique that preserves all information in the data set to be viewed, an important consideration for medical imaging data such as high-resolution MRI and CT data, as well as data from 3D Ultrasound scans. Chessboarding reduces the amount of extracted cell data that has to be moved from the server to the remote client by as much as 87%.

2. Background and Related Work

The isosurface visualization process can be configured as a pipeline, with a pipeline stage implementing each step of the isosurface visualization process. The first stage extracts the cells that intersect the isosurface from the raw data set or a transformed, search-optimized version of the raw data. The second stage (shown as data calculations in figure 1) determines how elements of the surface (usually triangles) pass through each extracted cell. The third stage performs rendering computations to account for such things as lighting, transformations, or texture mapping. The following stage prepares the 2-dimensional rendering of the isosurface. Finally, the scene is displayed on the screen, and the user may interact with the data (by issuing commands for rotating or zooming, for instance). Such commands may cause the rendering computations and surface computations to be repeated. With remote visualization, a programmer will divide the pipeline somewhere, deploying part on the server and the remainder on the client. An additional pipeline stage accounting for the data transport has to be introduced in-between the stages corresponding to the server and the client. We now discuss some recent work on remote visualization that have taken differing approaches in partitioning the isosurface rendering functions between the server and the client. Figure 1 shows the partitioning used in the schemes discussed.

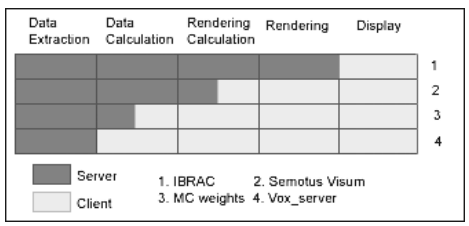

Figure 1: *Partitioning schemes for client and server.*

Luke and Hansen implemented the Semotus Visum system supporting multiple modes for remote isosurface visualization which minimized the processing and rendering burden on the client [LH02]. In one mode the server created a triangle mesh based on the color buffer and depth buffer. The mesh was sent, along with the rendered image, to the client. The client calculated texture coordinates based on the mesh, and applied the image as a texture map. In another, the server simply sent rendered frames. Experiments were run across a single hop on a LAN over 10, 100, and 1000 Mbps Ethernet. Engel, Westermann and Ertl proposed two alternative scenarios. In the first triangle strips are created to compress the geometry data before transferring it across the network [EWE99]. In the second, Marching Cubes [LC87] interpolation weights from each cell edge intersecting the isosurface are sent from the server to the client. Their server sent 26 bytes per cell in the most common case. This experiment was performed on a 100 Mbps LAN, and an in-memory octree was used to optimize the search for cells intersecting the isosurface. Yoon and Neumann proposed the IBRAC (Image Based Rendering Acceleration and Compression) scheme [YN00], where MPEG2-like compression of images of a 3D model are used. The client estimated the motion of pixels, and the server sent only those pixels that couldn't be estimated from its in-core, ray-traced 3D model.

Figure 1 also depicts the better of the two schemes we propose in this paper, the Vox_Server. The main distinction of our vox_server is that it uses cross-network pipelining to hide the cost of data computation on the client side, while sending the entire 3D model. Unlike IBRAC and Semotus Visum, this gives the user 360 degree interactivity with a lossless image. Unlike [EWE99], we use an out-of-core interval tree to optimize our search for cells. Further, we reduce the size and redundancy in the raw data, rather than in partially or fully processed data. We only need to send roughly 17 bytes per cell with out-of-core chessboarding (for 12 bit density values). With these features, we reduce the upfront cost to receive and display the 3D model.

3. Out-Of-Core Interval Trees

To allow the server to handle large data sets, well beyond the available RAM capacity, we transformed the raw data into a search-optimized data structure and stored it on the disk. Thus, the majority of the algorithm for out-of-core chessboarding lies in preprocessing the raw data cells.

The cell extraction technique used in this paper is a modification of the out-of-core interval tree implementation that was proposed in [SG98]. It is based on the memory resident data structure by [Ede80]and more recently used for fast in-core isosurface rendering [CMM*97]. The technique of [SG98] trades off disk storage for speed and permits the isosurface rendering of large data sets that cannot be held within the RAM. The detrimental effect of head seeking delays on the disk is reduced drastically by using an interval

tree layout that stores data in-line within the left and right lists of the disk-resident interval tree. The disk layout used permits the cells straddling the isovalue of interest to be accessed in an uni-directional scan of the disk, with minimal seeking across the lists of interest [SG98], [SG99]. The number of I/O requests as well as the delays introduced by head seeking are both minimized.

The average complexity of extracting an isosurface using interval trees has been shown to be O (log h + K), where K is the number of intervals intersecting the isosurface and log h is the height of the interval tree.

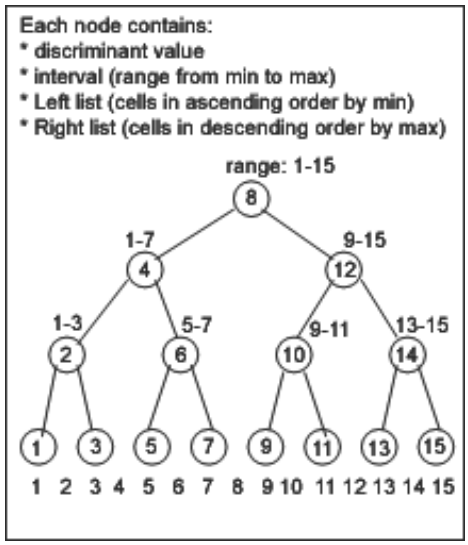

Figure 2: *Binary interval tree for isovalues 1-15. Cells consist of 8 sample points forming a cube. Cells in each node straddle the discriminant value and lie within its range.*

The creation of our out-of-core interval tree begins with a standard binary interval tree in linked list form. We then flatten out the structure into a pointerless array by storing the length of each node in a separate file, without compromising its optimal searching capabilities. The interval tree construction is a one-time pre-processing step. Once constructed, it can be used for all subsequent visualization of the data. The data layout of each subtree 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). Using the example in figure 2, the layout of nodes by discriminant value would be 8,4,6,7,5,2,3,1,12,10,9,11,14,13,15. This disk layout not only allows the out-of-core data to be accessed in query order but also allows a range of isovalues (which defines an isovolume) to be accessed in order with one pass. The data lists (see [Ede80] and [SG98]) within a node, including all cell information (density, gradient, location), are written in the right and left lists associated with a node (L_a, R_d) in order. To avoid having to load in all of a list's data or use random seeks, cell data is duplicated for both the L_a and R_d lists. Using the binary interval tree in figure 2, let's look at an example query of six. At the node labeled 8 (the root), we would scan the L_a list for cells whose minimum is less than or equal to 6. Then we would descend left to node 4 and scan the R_d list for cells whose maximum is greater than of equal to six. Next, we would descend right to node 6 and retrieve all the cells in the L_a list. Thus, we need to traverse 8,4, and 6, in the aforementioned order. Those are indeed the first three consecutive nodes in the file.

All list data is written sequentially to disk in binary format; no padding, pointers, or secondary file structures are used. The lengths of the intervals L_a (which equal the length of the corresponding R_d) for the nodes are stored in a separate "length" file. There are some notable advantages to storing the interval lengths on a separate file from the data lists. First, all interval lengths can be read into the RAM from the "length" file: this allows us to compute the number of disk blocks that have to be skipped during a search in advance, facilitating prefetching. Second, file access locality is improved, since the length and data files are streamed in independently. Unlike the k-ary interval tree of Chiang and Silva [CS97] we do not require any kind of searching within a node to determine the address of the next node to visit. We just add up node lengths and move directly to the node, reducing disk access time and disk storage in the process. Details of the out-of-core implementation are given in [SG99].

4. Out-Of-Core Chessboarding

For structured volume data sets, the storage requirement of out-of-core interval trees is large compared to the size of the raw data set. The process of transforming the raw data set into interval tree destroys the spatial coherence in the data. As a consequence, the cell's coordinates and sometimes gradients have to be stored explicitly within the out-of-core structure. In the modified out-of-core interval tree implementation used in this paper, we reduce the storage requirement of the out-of-core interval by adopting chessboarding scheme of Cignoni et al[CMM*97]. Although the concept of chessboarding used in our scheme is functionally not unique, there are several important implementation differences that arise from the need to support fast extraction using an out-of-core interval tree. These differences are summarized below and mainly stem from the fact that random accessing of the cell data using pointers is not possible with the disk-based out-of-core implementation.

- How cell data is stored:
  [CMM*97] stores cell data as the raw rectilinear data set within the RAM. The interval tree has pointers to cell data within its left and right lists. Out-of-core cell data is stored in-line within the left and right lists of the interval tree on the disk to avoid seeking delays.
- Additional data structures needed besides interval trees:
  For [CMM*97],only the raw data set itself is needed, ver-

sus hash tables to support chessboard computations in the order black cells are encountered during extraction.

- o Is any white cell data stored? How are gradients computed?
  For [CMM*97] the white cell data is stored as part of the original raw data set; this allows gradients to be computed on-the-fly. With out-of-core chessboarding, only data for black cells is stored. Gradients are precomputed and stored explicitly to avoid excessive disk seeking.
- o Order in which the surface going through the white cells are computed:
  In [CMM*97], the first time a white cell is encountered (while visiting a neighboring black cell) its surface is computed. With out-of-core chessboarding, the surface is computed only after all calculations for black cells are completed. It cannot be done in any other way if excessive disk seeking is to be avoided.
- o How are white cell intersections determined?
  [CMM*97] uses conditional checks of edges, whereas out-of-core chessboarding uses a lookup table, only in the interest of speed.
- o How is the Marching Cubes index calculated for white cells?
  [CMM*97] uses conditional checking of all vertices of a cell, versus using a vertex sign count and a lookup table (again in the interest of speed).
- o Applicability:
  [CMM*97] chessboarding is used in-core only; it cannot be naturally extended to handle out-of-core data sets. Our version can be used to compress both in-core and out-of-core interval trees.

The basic idea behind chessboarding is to recognize that the edges and vertices of cells are shared with neighboring cells. This sharing of edges implies that if a cell is intersected by an isosurface, the adjacent cells that share the cell's intersected edges must also be intersected by the isosurface. Thus with the proper organization, information does not need to be stored explicitly for every cell, resulting in a reduction of the overall storage requirement. To do this, chessboarding classifies cells as black or white cells. Each cell consists of eight data points at the eight corners of a cubic cell (or a rhomboid cell, for non-rectilinear data spaces). Information (such as pointers to the cells for in-core interval trees) only needs to be stored for black cells. All required information about white cells can be gathered from the neighboring black cells in all three dimensions. Chessboarding reduces the overall storage requirements of the disk-resident interval tree by a factor of four.

There are two major parts in the out-of-core chessboarding process. First, the raw data has to be written to file in as an interval tree without any white cells. Secondly, after traversing the on-disk tree and retrieving black cells, the white cell data that the isosurface passes through must be reconstructed.

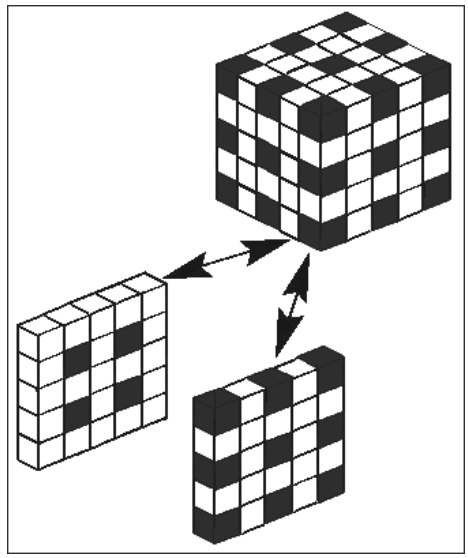

Figure 3: *Chessboarded cells in volume slices.*

The implementation of out-of-core chessboarding uses an array to hold **black triangle vertices**, and a hash table to hold information for adjacent white cells. A hash table is used since white cells get accessed repeatedly. A single white cell shares edges with multiple black cells, through which the isosurface might pass. The hash table facilitates fast access for updating a white cell entry. On the server side, the black cells are extracted by traversing the interval tree on the disk in the usual way. Extracted black cells are buffered on the server side and sent over to the client.

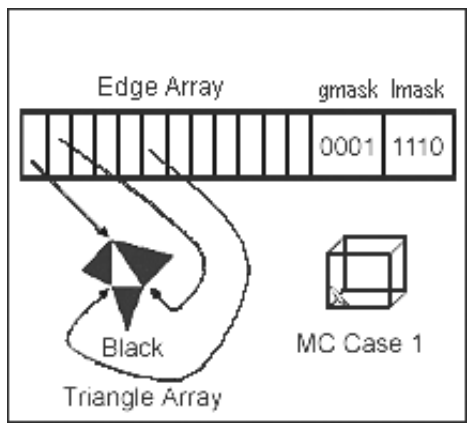

Figure 4: *White cell hash table entry.*

As each black cell is received by the client, its Marching Cubes lookup table index is determined, and black cell triangles are formed. Their vertex positions are immediately stored. The key issue is to then determine which white cells

would share an edge with the black cell and create or update its hash table entry. First we should mention that white cells are hashed based on the coordinates of the **origin of their cell**. We created lookup tables based on black cell marching cubes index which indicated **exactly which white cells will share intersecting edges with a given black cell**. As shown in figure 3, there are five white cells touching the front face of a black cell, eight white cells touching the middle, and another five white cells touching the back face of a black cell. We have three lookup tables representing three virtual slices, front, middle, and back, that surround a black cell. The key components of a table entry are the number of edges intersected for each white cell, followed by ordered pairs (white edge number and black edge number sharing an intersection). Pseudocode for creating the white cell entries follows a description of the last two fields of the white cell hash table entry.

The last two fields in the white cell hash table contain of a set of masks for greater than (gmask) and less than (lmask). The former indicates whether white cell vertices have values greater or less than the desired isovalue, and serve as an index for the white cell's Marching Cubes lookup. The latter (lmask), when OR'd with the gmask, confirms that all edges had been found (see Figure 4). The remainder of the entry holds pointers to the corresponding black triangle vertices to facilitate fast creation of white cell triangles; iterpolated black triangle vertex positions are re-used for white triangles.

```
  /* white cells origins, front slice */
static int front_origins[5][3]=
{{0, 7, 8},{3, 1, 8},{0, 4, 8},{6, 1,
8},{0, 1, 8}};

Look up black cell marching cubes
index

for (each cell origin in front_origins)
{
  Read in number of intersecting
  edge pairs from lookup table
  Hash white cell origin,
  return hash code


   /* set the black triangle vertex
 pointers for hash index */
 for ( each black-white
 intersecting edge pair )
 {
  Determine black triangle vertex
  pointed to.
  Point from hashed white cell edge to
  black vertex.
  Move on to next edge.
 }// for each intersecting edge pair
} // for each cell in front_origins

  Repeat for middle and back slices
```

5. Implementation

We implemented two versions of our application in order to test its feasibility. The first, called the triangle_server, extracted the cells off the disk, computed the triangles and sent them across the network to the client. The triangle_server did not use chessboarding on the disk-resident interval tree and was used only as a baseline. The second, called the vox_server, sent raw extracted black cells from a chessboarded interval tree on the disk and sent them to the client.

5.1. Common Features

Both versions had the following features in common:

- o out-of-core interval tree for optimized cell extraction
- o Marching Cubes algorithm to create polygons
- o OPENGL and GLUT API for rendering and display
- o producer-consumer with bounded buffer

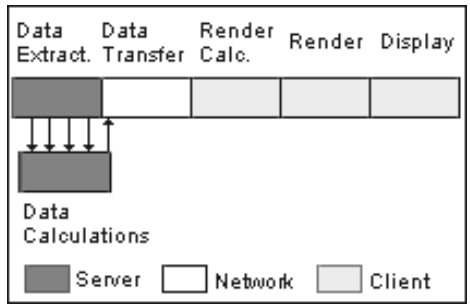

Figure 5: *Triangle_server remote visualization pipeline*

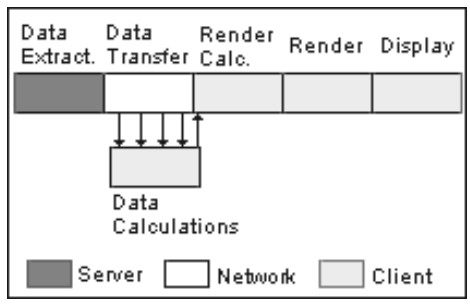

Figure 6: *Vox_server remote visualization pipeline*

The producer-consumer with bounded buffer algorithm was used to add concurrency within the remote visualization pipeline. The consumer(s) were responsible for creating the

triangles that made up the isosurface. In the case of the triangle_server, both producer and consumers resided on the server. Computation of triangles was concurrent with file I/O, and hid the performance cost for seeking and reading. Moreover, the triangle_server gained additionally concurrency with multiple consumer threads. In the second version, the vox_server acted as the producer, and the client was the consumer. This meant that data transfer was concurrent with triangle calculation on the client side. While the client consumer performed Marching Cubes on black cells and created white cell triangles, the server continued to put data into flight on the network. Figure 5 and figure 6 show the remote visualization pipeline for the triangle_server and vox_server respectively. Note that the additional stage of data transfer across the network is depicted explicitly.

5.2. Implementation Details

The client and server used TCP sockets for reliable delivery. The visualization process started with the client sending a query to the server. The query included parameters such as number of threads, number of buffers, number of blocks per buffer, and number of triangles to be sent in a chunk. We tried to maximize the latter to reduce the number of send requests. (It is faster to send a few large packets than many small packets, and we also wanted to reduce the number of system calls). Since we ran the server program on a two processor node, the triangle_server used two consuming threads and three buffers. Both client/server pairs used blocking sockets. One extra send was needed in advance to notify the triangle_client of the number of triangles to expect; the vox_server and client simply agreed on a send size.

The vox_client sent exactly the same type of request as the triangle_client. There was a small difference, however, in how the parameter for number of blocks per buffer was used. Since the vox_server sent raw data, the buffer size became the requested send size. We used a size of 22 Kbytes, the same buffer size as was used with the triangle_server. Since the vox_client and vox_server worked in total isolation from each other, normally shared global data structures were not immediately accesible on the client. File metadata such as cell size, dimensions of the volume, and bits per density were sent ahead to enable the client to recreate the global structures. Additionally, statistics on the number cells per isovalue were sent to help the consumer determine the size of the white cell hashtable. Sending these had a one-time cost of 32 KBytes. (Note that although the size of the statistics set grows with the size of the data set, it is still negligible next to the amount of actual data to be sent.)

6. Results

We ran both the multithreaded triangle_server and the sequential vox_server from a cluster node containing two 2.2 GHz Pentium 4's sharing 4 gigabytes of RAM and reading from the disk via a 15,000 RPM SCSI drive. The client was a Dell Dimension 4500 with 2.0 Ghz Pentium 4, 512 MB RAM, 64 MB GeForce4 MX graphics card and 100 Mbps Ethernet card. The tests were conducted with rectilinear CT scan data. It was comprised of the first 256 slices of the Visible Woman data set (512x512x256) where each slice was 200 data points in thickness. The Visible Woman used 12 bits per density value (isovalues 0-4096). The size of the raw data set was 128 MB. When converted to a binary interval tree, the size was 6.7 GB, and the chessboarded interval tree size was 1.7 GB.

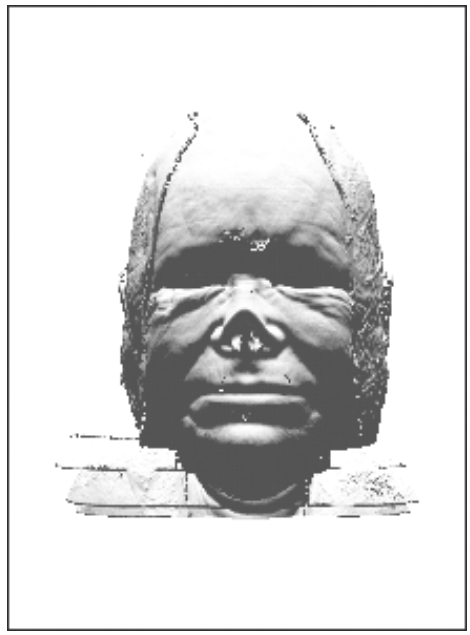

Figure 7: *First 256 slices of the Visible Woman data set.*

In order to minimize variance in network traffic conditions, the measurements were taken alternating between vox_server and triangle_server between 11 a.m. and 8 p.m. EST. We tested the total time from extraction off the disk to receipt of triangles for every 100th isovalue.

Initial tests were across several switches on a 100 Mbps switched LAN. The first test was simply to compare the number of bytes sent by the triangle_server versus the vox_server. For the majority of query isovalues, the vox_server sent one-eighth the amount of data that the triangle_server needed to send. For query results with below 1000 triangles, the vox_server sent more data since it had to send black cells from the exterior faces of the volume under all circumstances. The majority of the query isovalues returned results in the 97,000 Kbyte to 206,000 Kbyte range as triangles with a peak return of 479,679 Kbytes. Returning black cells for the same queries gave a midrange of 11,718 Kbytes to 25,585 Kbytes and a peak value of 58,899 Kbytes.

We next measured the time. For the majority of isovalues, the vox_server was 68% faster than the triangle_server, and displayed an isosurface on the screen in under 8 seconds. Roughly two-thirds of the time was spent on the vox_client side, processing raw cells into triangles. Remote visualizations times were extremely fast in this case, as server side delays and transport delays were effectively overlapped. We did find, at the peak isovalue, evidence of thrashing on the client side due to the size of the white cell hash table. We believe this can easily be resolved at the preprocessing stage by creating several chessboarded interval tree files. The client would then request the files one by one, recreating a hash table each time. Thus, with a little additional communication, this technique could achieve even greater scalability.

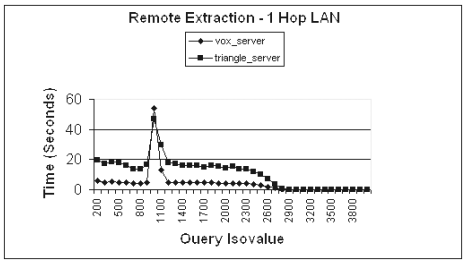

Figure 8: *Time to receive data and display an isosurface via LAN.*

Figure 9 shows a comparison between the two implementations on a stage-by-stage basis for a typical isovalue (1400 in this case). The triangle_server is clearly faster at computation of triangle vertices and normals, due to multithreading on the dual processor. File I/O takes much longer for the triangle_server since it must do four times as much reading, and four times as much seeking. Seek times have not improved at the same rate as data transfer rates, so a factor of four increase in seek distance will mean much greater total file I/O time. Still, with the multiple threads, much of the triangle_server's file I/O time can be hidden by concurrency with computation.

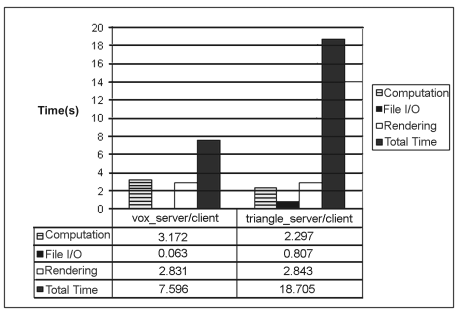

Figure 9: *Stage-by-stage comparison between vox_server and triangle_server for a typical isovalue.*

Another interesting result shown in the figure is that excepting rendering, 66% of the total time for the vox_server and client is used for computation. Although the low total time can be partially attributed to sending less data, data transmission and computation are clearly occurring concurrently on the client. Moreover, the vox_server's pipelining occurs during the two longest stages, the most advantageous pair of stages to overlap.

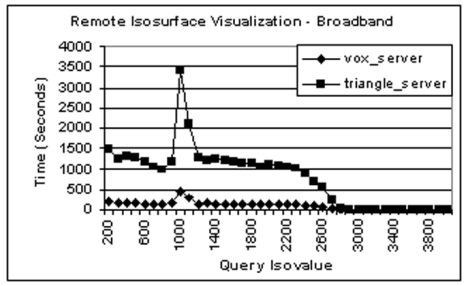

Figure 10: *Time to receive data and display an isosurface via broadband.*

The final set of tests were run over commercial cable. The cable wire provided 2 Mbps bandwidth. A route trace showed that our data traveled 5 hops over broadband plus 1 hop across the LAN. The time to receive data increased from on the order of seconds to minutes. In this case, the network delay dominated the other phases of visualization and network delays could thus not be hidden with any kind of overlapping of server-side functions and transport functions. The median vox_server time in this case was close to 2 minutes, and the peak time was 7 minutes. At this point computation dropped to 1.7% of total time. Processing of data could be totally hidden by the latency of the network. Here the vox_server was 87% faster than the triangle_server, illustrating that computation costs were totally hidden by network latency.

7. Discussion and Conclusions

Our experiments have shown that the ratio of black cell data sent to triangle data sent is 12.5%. Likewise, we have found a reduction in time to receive and view an isosurface of roughly 87%. To understand why this is happening, we need to re-examine the marching cubes algorithm. Suppose that a cell and a triangle were about the same size. (In actuality, a Visible Woman cell is 68 bytes, slightly smaller than our triangle of 72 bytes.) In all but one case, when an isosurface passes through a cell, two or more triangles are produced. By sending cells instead of triangles we necessarily reduce the amount of data sent by 50%. Further, with chessboarding, we carry one-fourth the number of cells we would normally. 100% *.50 *.25 = 12.5%. For eight bit data, the results clearly would be even better.

The vox_server design presented in this paper allows isosurfaces to be viewed quickly on remote clients. The lossless compression achieved using out-of-core chessboarding helps significantly in this respect. With the advent of the multi-gigahertz processors and increased bus speeds for the servers and clients, there is clearly no disadvantage to shift some processing to the client side; the dominating factor is now network latency. On fast networks, server side processing time and network transport time can be effectively overlapped to hide the network delay. On relatively slower networks, the overall visualization delay is determined by the network performance. Sending chessboarded cells across the network shows a positive advantage and potential as a technique for data reduction and faster access to remote data for visualization for both fast and slow networks. The vox_server design thus facilitates such things as remote diagnosis and collaborations, which are likely to use faster networks (like the Internet 2) for connecting two widely-separated sites or the fast local network within an enterprise (such as a hospital).

References

[CMM*97] CIGNONI P., MARINO P., MONTANI C., PUPPO E., SCOPIGNO R.: Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics 3*, 2 (1997), 158–170.

[CS97] CHIANG Y., SILVA C.: I/o optimal isosurface extraction. In *Proc. of Visualization '97* (1997), pp. 293–300.

[Ede80] EDELSBRUNNER H.: *Dynamic data Structures for Orthogonal Intersection Queries*. Technical Report F59, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980.

[EWE99] ENGEL K., WESTERMANN R., ERTL T.: Isosurface extraction techniques for web-based volume visualization. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)* (1999), IEEE Computer Society.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987), ACM Press, pp. 163–169.

[LH02] LUKE E. J., HANSEN C. D.: Semotus visum: a flexible remote visualization framework. In *Proceedings of the conference on Visualization '02* (2002), IEEE Computer Society, pp. 61–68.

[SG98] SULATYCKE P., GHOSE K.: Out-of-core interval trees for fast isosurface extraction. In *Proceedings of Late Breaking Hot Topics IEEE Visualization '98* (1998), pp. 25–28.

[SG99] SULATYCKE P., GHOSE K.: A fast multithreaded outofcore visualization technique. In *Proceedings of International Parallel Processing Symposium* (1999), pp. 569–575.

[YN00] YOON I., NEUMANN U.: Webbased rendering with ibrac (image based rendering acceleration and compression). In *Proc. of Eurographics* (2000).