LETTER

Hybrid Parallel Extraction of Isosurface Components from 3D Rectilinear Volume Data*

Bong-Soo SOHN[†a)], ***Member***

SUMMARY We describe an efficient algorithm that extracts a connected component of an isosurface, or a *contour*, from a 3D rectilinear volume data. The efficiency of the algorithm is achieved by three factors: (i) directly working with rectilinear grids, (ii) parallel utilization of a multi-core CPU for extracting active cells, the cells containing the contour, and (iii) parallel utilization of a many-core GPU for computing the geometries of a contour surface in each active cell using CUDA. Experimental results show that our hybrid parallel implementation achieved up to 20x speedup over existing methods on an ordinary PC. Our work coupled with the Contour Tree framework is useful for quickly segmenting, displaying, and analyzing a feature of interest in 3D rectilinear volume data without being distracted by other features.

key words: *isosurface, contour tree, contour propagation, multi-core*

1. Introduction

Isosurface extraction is one of the most popular techniques for visualizing a volumetric dataset. An isosurface is a level set surface defined as $I(w) = \{(x, y, z)|F(x, y, z) = w\}$, where F is a scalar field data and w is an isovalue. Since a single isosurface may consist of many connected components, or *contour*s, with each representing a different feature, the ability to quickly extract an individual isosurface component is critical for finding and analyzing a desired feature of the data without being distracted by other features (Fig. 5).

The Contour Tree (CT) is useful for visualization of contours [6], [9]. CT provides a topological structure of contours, and it is used as an interactive user interface to segment and display individual contours. CT also generates seed cells for efficient extraction of the contours. The extraction process uses a contour propagation algorithm that incrementally finds and triangulates active cells that are connected to a given seed cell through contour surfaces. We compute CT and seed sets from input volume data as preprocessing and use them to interactively select and extract individual contours. This process is depicted in Fig. 1.

The performance of the contour propagation algorithm is critical for obtaining necessary interactivity in the CT interface. However, there are two efficiency problems in existing approaches for contour propagation. (i) When applied to rectilinear volume data, the propagation from one seed cell may generate two or more contours because a cube may contain more than one contour [3]. Previous methods [6], [9] subdivided each cube into five or six tetrahedra to avoid the incorrect result. However, the decomposition of each cell results in significant degradation of efficiency because it increases the number of cells by five or six times. (ii) The propagation methods have not been parallelized and do not utilize parallel computing resources of ordinary PCs.

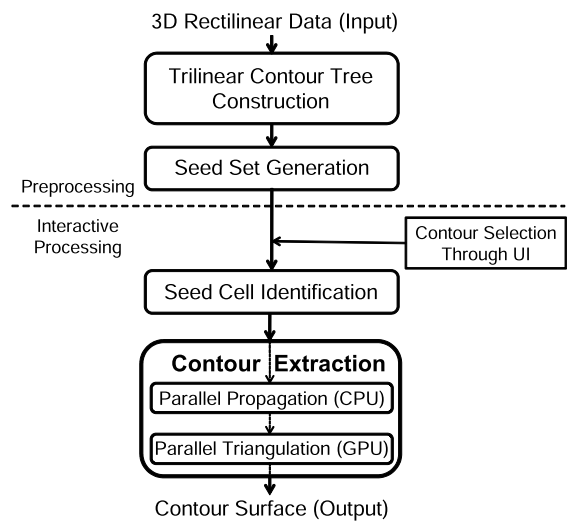
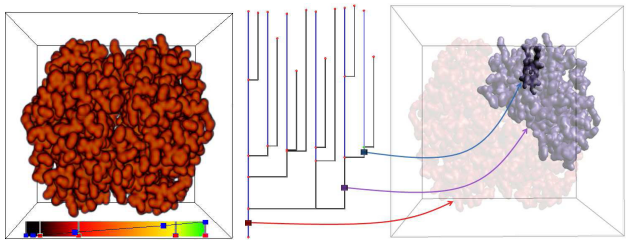


Fig. 1 Overall process for parallel contour extraction.

In this letter, we describe an efficient algorithm that selects and extracts individual contours from 3D rectilinear volume. We designed our propagation algorithm to directly work with rectilinear data without any subdivision and generate a single topologically correct contour that intersects with a given seed cell. This approach reduces the number of active cells, and hence improves the speed of contour extraction and reduces the size of resulting contour meshes.

We further enhance the performance via parallel utilization of multi-core CPUs and many-core GPUs that are common in ordinary PCs. The number of cores in current CPUs ranges from two to eight, while current GPUs have hundreds of cores. Current trends in CPU/GPU technology indicate that the numbers of cores are expected to increase, rather than clock speeds [5]. Therefore, the design of a parallel algorithm that efficiently utilizes multi-core CPUs and many-core GPUs is critical for improving performance. We design a parallel algorithm for contour propagation that utilizes a multi-core CPU. We also separate a data-parallel part (triangulation step) from the propagation process to opti-

Manuscript received January 24, 2011.
Manuscript revised August 4, 2011.
[†]The author is with the School of Computer Science and Engineering, Chung-Ang University, Seoul, Korea.
*This research was supported by the Chung-Ang University Research Grant in 2010.
a) E-mail: bongbong@cau.ac.kr
DOI: 10.1587/transinf.E94.D.2553



(a) Volume Rendering

(b) Contour Extraction using Contour Tree User Interface

Fig. 2 CT is constructed from an approximate electron density volume of a hemoglobin molecule (a). Using CT interface (b), we selected and extracted individual contours of the whole molecule (red), a polypeptide subunit (purple), and a heme structure (dark blue).

mize the part via GPU acceleartion. Experimental results indicate that we achieved up to 20x speedup over the previous method [6] that performs single-threaded propagation from tetrahedral grids to extract a selected contour.

2. Contour Tree and User Interface

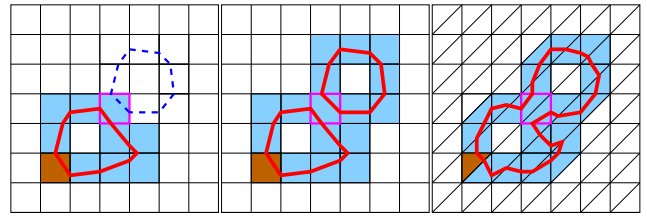
Consider a scalar field F defined from the input volume data. As an isovalue changes from the maximum to minimum value, contours are created, merged, split, and destroyed. CT represents the topological events in the form of a tree. Each vertex of CT represents a critical point of F where the topological events occur. Each edge of CT represents a maximal set of continuous contours that do not contain critical points. When CT is laid out on a 2D plane, vertical coordinates of CT vertices are determined as the function values on the corresponding critical points. A horizontal coordinate of a CT vertex is determined according to the number of vertices in the subtree that is connected to the CT vertex.

A seed set of CT edge is defined as a subset of edges on the input domain mesh where any contour that is associated with the CT edge intersects with an edge in the seed set. A cell that contains the intersecting edge becomes a seed cell for extracting the contour. We compute and store a seed set for each edge of CT in a preprocessing step. We refer to [8] and [6] for details of CT concepts, CT construction, and seed set generation (pathseed method).

A point on a contour tree interface has one-to-one correspondence to a contour. Users may select and extract a specific contour by clicking a point on the contour tree. When a desired contour is selected, a seed cell is identified from the precomputed seed set and the entire surface of the selected contour is efficiently extracted from the seed cell using our propagation method (Fig. 2).

3. Proposed Approach for Contour Extraction

The original propagation algorithm [3], [6] initially inserts a seed cell into an empty queue. The seed cell is propagated through active cells by repeating the process of dequeuing and triangulating a cell, and then enqueuing neighboring cells that are connected to the cell by contours in the cell until the queue becomes empty.



(a) Our method

(b) Propagation on rectilinear grids

(c) Propagation on tetrahedral grids

Fig. 3 Comparison of our propagation method and previous methods.

We divide the original algorithm into two separate steps: (i) a propagation step that identifies a set of active cells that contain the selected contour, and (ii) a triangulation step that computes the geometries (e.g., vertex coordinates, normals, and connectivity) of the contour surface in each active cell. The division is necessary because the intensive work of interpolation in the triangulation step can be highly optimized through CUDA programming that fully utilizes the many-core GPUs, while the propagation step is hard to optimize using many-core GPUs due to its incremental and data-dependent algorithm.

3.1 Contour Propagation from Rectilinear Data

Since a cube in rectilinear data may contain more than one contour, the propagation from a seed cell may generate two or more contours, as shown in Fig. 3 (b). In order to avoid the incorrect result, we consider only the selected contour and ignore undesired contours when processing an active cell during propagation, as shown in Fig. 3 (a). For this purpose, we assign an id number (e.g., 0, 1, 2, or 3) to each contour in the cell and associates the cell with the id number of the selected contour. A cell containing the selected contour is represented as a pair (cell number, contour id number). Since most active cells have a single contour in practice, the overall overhead for processing the cells that have more than one contour is insignificant. Each cell in Algorithm 1 represents the pair (cell number, contour id number). When ambiguity exists for triangulating a cell with Marching Cubes table, we apply a disambiguation method [7]. For such cells, the triangulation is performed during the propagation step as exception because CUDA kernel function is optimized for dealing with only simple cases that involve no ambiguity.

3.2 Parallel Propagation Algorithm

The basic structure of the propagation algorithm is based on a FIFO queue. The algorithm has a loop that repeats taking a cell c from a queue, processing c, and inserting new cells to a queue. We parallelize the algorithm by simultaneously executing the loop with multiple CPU threads. We use a variant of work-stealing methods [4] where each thread maintains a lock-free local queue. In the method, a thread can steal a work (cell) from another thread for dynamic load balancing when a local queue is empty. This distributed task queue

method is suitable for achieving high scalability in our multithreaded and shared-memory environment because using local queues is designed to be safe in the algorithm and the synchronization overhead is minimized compared to the centralized method that uses one global shared queue.

The pseudocode of our parallel propagation algorithm is shown in Algorithm 1. The basic structure is the same as that of the serial algorithm except that multiple CPU threads execute PROPAGATE() simultaneously. In PROPAGATE(), each thread maintains a lock-free local queue. We also introduce a lock-protected global queue through which a thread that has an empty local queue can steal a cell from another thread. If a local queue is empty, then that information is announced to other threads through *wait_flag* array and a new active cell is inserted into the global queue instead of the local queue in another thread. Then, the thread that has the empty local queue takes the cell from the global queue. Since the access to the global queue occurs very rarely in practice while lock-free local queues are used most of the time, the synchronization overhead related to queue accesses is minimized during simultaneous propagations.

Algorithm 1 Multithreaded contour propagation algorithm

```
PARALLELEXTRACT(s , thread_no)
Input - s : seed cell , thread_no : the number of threads
Output - A list of active cells that contain the desired contour
Visit(s);
Global.Enqueue(s);
for i ← 1 to thread_no − 1 do
    CreateThread(PROPAGATE(i , thread_no)); // run in parallel
Join(); // waits until all threads call thread_exit().

PROPAGATE(thread_id , thread_no)
while (true)
    c ← DEQUEUE();
    if (c == EMPTY) then continue;
    InsertActiveCell(c);
    foreach face f_i of c that contains a contour
        c′ ← a cell sharing f_i with c;
        lock(c′);
        if (!isVisited(c′)) then
            Visit(c′);
            unlock(c′);
            ENQUEUE(c′);
        else
            unlock(c′);

DEQUEUE()
// If local queue and global queue are empty, return EMPTY
c ← Local.Dequeue(); // dequeue from local queue
if (c == EMPTY) then
    c ← Global.Dequeue(); // dequeue from global queue
    if (c == EMPTY) then
        wait_flag[thread_id] = true;
        if (all wait_flags are true) then thread_exit();
        return EMPTY;
    else
        wait_flag[thread_id] = false;
return c

ENQUEUE(c)
if (any one of wait_flag is true) then
    Global.Enqueue(c); // enqueue c to global queue
else
    Local.Enqueue(c); // enqueue c to local queue
```

3.3 Parallel Triangulation Using CUDA

After the list of active cells is computed in multithreaded CPU mode, we triangulate each active cell through a GPU using CUDA. To optimize the implementation of triangulation, we modified a CUDA kernel function `generateTriangles2` that is found in nVidia's Marching Cubes sample code [1]. Each GPU thread generated from CUDA takes an active cell from the list and computes the vertex positions, normals, and connectivity of contour surfaces in the cell based on Marching Cubes triangulation tables. The computed geometries are written into an output array that stores the triangles of the selected contour. The location in the output array must be precomputed for each active cell so that the CUDA kernel function directly writes the triangles into the pre-determined position. This is necessary to avoid communication among GPU threads.

4. Experimental Results

We implemented our algorithm and tested it on a desktop PC equipped with an Intel i7 (2.66 GHz) CPU with four cores, 3 GB main memory, and an nVidia GeForce 480GTX graphics card that has 1.5 GB video memory.

We tested two datasets, a head MRI that highlights the Cerebro-Spinal-Fluid filled cavities [2] and an approximate electron density map of a hemoglobin molecule. Figure 2 and 5 show the results of extracting a polypeptide subunit contour from the hemoglobin dataset and a ventricle contour from the head MRI dataset, respectively. CT computed from each dataset was simplified before display based on [6] because the original trees contain too much complexity. As shown in Fig. 5 (c) and (d), an isosurface of the head MRI dataset contains many contours that represent brain tissues and eyeballs as well as the ventricle. Using our method, we extracted only the ventricle contour that is the main feature of the data. This allows us to visualize and analyze the ventricle surface without being distracted by other features.

We measured the contour extraction times using the previous method [6] and our method with different numbers of threads. The previous method that was compared with our method is Carr et al.'s method described in the paper [6]. In the previous method, rectilinear grids were converted to tetrahedral grids and then single-threaded contour propagation was performed on the tetrahedral grids to extract the selected contours. We used the pthread library to create CPU threads and measured the time with and without GPU acceleration. The results are summarized in Table 1. The graphs in Fig. 4 analyze the results in Table 1. As shown in the graphs, our basic method using a single thread with no GPU utilization achieved 2x-3x speedup over the previous method [6]. The performance of our parallel implementa-

Table 1 Timing results for contour extraction algorithms with different numbers of threads. FPS (Frames Per Second) is computed as 1/(extraction time). In the previous method (prev) [6], single-threaded propagation was performed on tetrahedral grids for contour extraction.

data / size	GPU	prev [6]	our method (FPS)			
		1	1	2	3	4
Hemoglobin 256x256x256	*with*	3.1	4.7	8.9	13.0	16.9
	without	0.8	2.5	4.7	6.9	9.0
Head MRI 256x256x128	*with*	5.6	8.5	16.3	23.8	31.0
	without	1.8	4.7	8.7	12.5	16.2

Table 2 The comparison of the number of active cells and the number of triangles generated for the selected contours.

data	previous method [6]		our method	
	cell#	*tri#*	*cell#*	*tri#*
Hemoglobin	1,171,571	1,439,588	245,485	491,130
Head MRI	523,767	644,704	108,868	224,456

Table 3 Timing results for the propagation step (single CPU thread) and the triangulation step (with or without GPU acceleration). (unit : second)

data	propagation (*CPU*)	triangulation	
		without GPU	*with GPU*
Hemoglobin	0.208	0.191	0.004
Head MRI	0.114	0.098	0.002

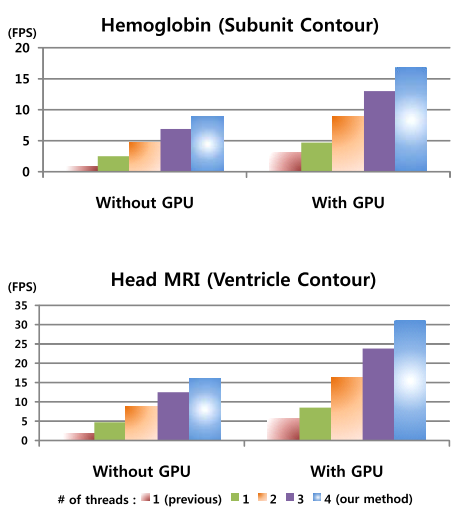


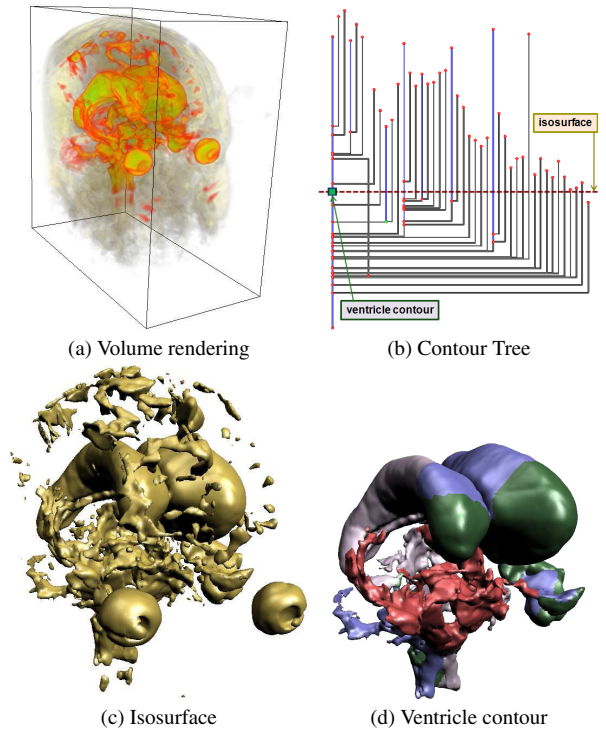
Fig. 4 Performance of our parallel algorithm for contour extraction.

tion is also shown in Fig. 4. We achieved up to 20x speedup of contour extraction with high scalability.

Table 2 shows that the number of active cells in our method is 4 - 5 times smaller than the number in the previous method [6]. This improves the performance and reduces the contour mesh size. Table 3 shows the time required for triangulation with and without GPU utilization. The result indicates that the computation required for triangulation is very suitable for GPU acceleration and that the GPU implementation led to significant improvement in performance.

5. Conclusion

We described a hybrid parallel algorithm that efficiently extracts an isosurface component from 3D rectilinear volume data. High performance was achieved via optimized parallel utilization of multi-core CPUs and many-core GPUs that are common in ordinary PCs. Our work coupled with the Contour Tree framework is useful for quickly finding and displaying a feature of interest in volume data.



(a) Volume rendering (b) Contour Tree

(c) Isosurface (d) Ventricle contour

Fig. 5 The result of extracting ventricle contour from a head MRI data (a). An isosurface (c) consists of many contours. The ventricle contour surface (d) that is a main feature of the data is efficiently segmented using our parallel propagation method and contour tree interface (b). Colors in (d) show contribution of four different CPU threads.

References

[1] nVidia CUDA SDK Code Samples: Marching Cubes. http://developer.nvidia.com/object/cuda_sdk_samples.html
[2] Real world medical datasets. http://www.volvis.org
[3] C.L. Bajaj, V. Pascucci, and D.R. Schikore, "Fast isocontouring for improved interactivity," Proc. VolVis, pp.39–46, 1996.
[4] R.D. Blumofe and C.E. Leiserson, "Scheduling multithreaded computations by work stealing," J. ACM, vol.46, no.5, pp.720–748, 1999.
[5] S. Borkar, "Thousand core chips - a technology perspective," Design Automation Conference, pp.746–749, 2007.
[6] H. Carr, J. Snoeyink, and M. van de Panne, "Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree," Computational Geometry: Theory and Applications, vol.43, no.1, pp.42–58, 2010.
[7] G.M. Nielson, "On marching cubes," IEEE Trans. Visualization and Computer Graphics, vol.9, no.3, pp.283–297, 2003.
[8] V. Pascucci and K. Cole-McLaughlin, "Parallel computation of the topology of level sets," Algorithmica, vol.38, no.2, pp.249–268, 2003.
[9] B.-S. Sohn and C. Bajaj, "Time-varying contour topology," IEEE Trans. Visualization and Computer Graphics, vol.12, no.1, pp.14–25, 2006.