International Journal of Image and Graphics

Parallel Rendering for Legible Illustrative Visualizations of Dense Geometries on Commodity CPUs

Haipeng Cai
Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, Indiana 46556, United States
hcai@nd.edu

Received (26 June 2015)
Revised (21 September 2015)
Accepted (XX XX XXXX)

This paper presents a parallel visualization technique for illustrative rendering of dense three-dimensional (3D) geometry data sets. Our approach maps the depth information in each geometry onto various visual dimensions of graphical representations, including shape, color, brightness, transparency, and size, to achieve legible display in dense geometry environments where visual clutters often hinder perception and navigation in the visualizations. At the same time, we leverage legacy CPU computing power to overcome performance challenges as a result of the depth-dependent illustrations used for the visual legibility enhancement. This is realized by a novel parallel rendering algorithm we developed particularly for illustrative visualizations of depth-dependent stylized dense geometries at interactive frame rates. While the computation could be performed atop modern GPU devices, we target a parallel visualization framework that enables it to efficiently run on commodity CPUs, which are much more available than GPUs for ordinary users. We evaluated our framework with visualizations of depth-stylized geometries derived from 3D diffusion tensor MRI data, by comparing its efficiency with several other alternative parallelization platforms with respect to the same computations. Results show that our approach can efficiently render highly dense 3D geometry data sets and, thus, it offers not only an alternative and complementary, but also more adoptable, solution to users in contrast to parallel visualization environments that rely on GPUs.

Keywords: Illustrative visualization; parallel rendering; 3D geometry; depth-dependent rendering; stylized visualization; visual legibility

1. Introduction

When visualizing large-scale geometrical data such as dense tubes or simply polylines, one of the critical issues lies in the visual perception in the depth dimension due to inherent clutters or occlusions as a result of overlapping graphical objects and/or structures. To improve the overall visual legibility of three-dimension (3D) data visualizations, mapping depth information to various visual variables of graphical representations can be an effective means for enhancing depth perceptions, especially during user interactions in the context of interactive visualizations.

On the basis of the semiology of graphics [13], various visual variables, such as size, color, brightness (value), and transparency, etc., can be used for the depth mappings. For

instance, a linear mapping from per-vertex depth value to the radius (i.e., the size) of a tube in a visualization of dense 3D geometries can give users a visual cue for discerning depth positions as the radii monotonically decreases along the viewing direction. Similarly, a consistent mapping from depth to color provides a constant correlation between the view distance from the viewer to the geometries and the color, thus it helps the viewer navigate the visualized data set along the depth direction. In both cases, better depth perception can be obtained to enhance the overall legibility of the rendering in the visualization.

To render the depth-dependent visualizations at interactive frame rates, real-time computation involved in the depth mappings is desirable and usually required. In that regard, there are two major computation-intensive steps to be performed every time depth reordering is needed—for example, the data view is changed as a consequence of rotating the visualization. First, depth values are calculated according to updated viewing directions and sorted along those directions. Second, mappings are computed and the geometries are rendered over again to update the visualization. Concisely, depth sorting and re-rendering should be performed once depth order is changed due to view transformations which disrupt the visual distance from users to individual geometries. To obtain an interactive frame rate of such visualizations, therefore, these computations need be performed in a real-time fashion, which have been proven difficult either with sequential approaches or by direct use of *general-purpose* parallel-computing facilities.

To address such performance challenges, it is reasonable to consider parallelizing the depth-dependent visualizations for interactive rendering. While GPUs are being increasingly applied in many modern parallel computations and, indeed, visualizations of large-scale dense geometry data could be a perfect fit for GPU computing platforms, we aim at a cheaper solution to the same challenges. In particular, we target a solution that can be a useful complement to the GPU computing paradigm, especially when GPU devices and related high-end hardware configurations are not readily available. In fact, this is mostly true since GPUs are generally much more expensive than ordinary computer users would like to afford for tasks like visualizing dense geometries.

In this context, we present a parallelized illustrative visualization approach that enables real-time computations for interactive depth mappings hence enhanced visual legibility in 3D visualization environments. Our technique utilizes the message passing interface (MPI) in collaboration with the Visualization ToolKit (VTK) [16] while extending current VTK facilities for the purpose of performance optimization. Through the optimized coordination between a parallel depth ordering algorithm and parallel rendering method with customized data structures for real-time depth mappings, our approach has been shown as an efficient solution in visualization use scenarios involving dense 3D geometry data sets.

We applied our approach to depth-dependent 3D dense geometry visualizations with depth mapped to primary visual variables, and have obtained interactive rendering speed with either single variable mapping applied or multiple variable mappings combined. It is noticeable that even with the combination of mappings from depth to size and those to any other visual variables, in which two passes of depth sorting and rendering plus geometry generation from polylines are all required for each frame, our approach has still been able to render the dense data sets at interactive frame rates.

The main contributions of this paper are the legibility-enhanced visualization of dense 3D geometries and the *CPU-based* parallel rendering scheme. While we demonstrate its application only in context of illustrative visualizations of geometry data, the parallel rendering can also be applied to relevant other performance-critical scenarios where, for example, real-time sorting of vertices is needed.

The rest of this paper is organized as follows. Section 2 highlights the legibility issues encountered in our visualization scenarios with existing approaches that motivate this work and gives necessary background necessary for understanding our approach and experimental data set. Then, we describe the details about our approach in Section 3 and key implementation issues in Section 4. Section 5 presents our empirical studies and evaluation results, followed by an extended discussion on the implications of those results in Section 6. We discuss previous research related to our approach in Section 7 and give concluding remarks finally in Section 8.

2. Motivation and Background

This work was originally motivated by our research on visualizing the diffusion tensor magnetic resonance imaging (DT-MRI or DTI) data sets [6,4]. As an advanced MRI technique, DTI has been shown advantageous over other imaging techniques in enabling *in vivo* investigation of biological tissues. Specifically in our work on scientific visualizations targeting neuroscientists and radiologists, we are primarily concerned about the DTI model of human brains. One way to visualize the brain DTI data sets is to, via 3D tractography, reconstruct the geometrical model of the distribution and connectivity of neural pathways in the brain white matter. For our end users (i.e., the domain scientists), these pathways represented by 3D geometries (e.g., polylines and tubes) can greatly assist with understanding the internal structure of brains hence diagnosing cerebral anomalies and planning neurological surgeries [4,7].

However, under a common data-acquisition setting (e.g., a scanning resolution of 0.9375mm x 0.9375mm x 4.52mm), the resulting 3D DTI geometry model usually contains over 10,000 polylines each consisting of up to 100 line segments. What is more challenging is that not only is the scale of these geometries quite large, but also they are extremely dense, reflecting the cramped layout of neural pathways in human brains. Consequently, directly rendering such highly dense geometries in a 3D setting ends up with visualizations full of visual clutters [6], making it quite difficult for domain scientists to explore hence understand the brain model.

Moreover, streamtubes [8] are usually more preferable than polylines as they are able to convey more data characteristics, such as shape, size and orientation, of the brain neural pathways. Unfortunately, streamtube visualizations tend to suffer much more greatly from legibility issues such as visual clutters than polyline representations due to the need for expressing the added data properties. In fact, previous studies have shown that even reaching a region of interest in a dense streamtube visualization appears quite difficult even with aids of specialized interaction facilities [7,5]. Despite of a large body of previous research on visualizing DTI data sets [4], existing approaches mostly turn to render the data

sets with different metaphor or graphical representations (e.g., visualizing the underlying tensor field itself or showing only higher-level abstractions of the data model), missing the benefits of facilitating the comprehension of the spatial structures and internal connectivity of the brain.

To address the legibility issues in the 3D visualization of dense data set, an important and effective approach is to enhance depth perception as the loss of depth information accounts largely for the difficult navigation in 3D rendering environment [14,26]. And depth mapping to various visual elements can greatly alleviate that difficulty as it provides additional visual cues that help viewers orientate themselves along the depth dimension. On the other hand, however, such mappings can also substantially increase the overhead of rendering, which is particularly true in interactive visualizations as in our situation. While the modern GPU computing architecture is designed to address challenges in graphics rendering performance, the commodity CPUs are more commonly seen in legacy computing platforms. Thus, a much cheaper solution that provides real-time visualizations with depth-enhanced illustrations seems to be right in demand.

3. Approach

3.1. *Depth Dimension Management*

Visual legibility of two-dimensional (2D) graphical representations can be characterized by graphical density, angular separation, and retinal separation [13]. Further, retinal separation is defined by six visual variables: size, color, shape, value, orientation and texture. Motivated by the legibility rules defined in terms of these dimensions, we explore the visual legibility issues in 3D data visualizations by examining 3D legibility dimensions. While our exploration is still based on the 2D legibility framework, certain expansion is required to characterize legibility in 3D environments. To that end, we expand that framework from 2D to 3D by adding the depth-separation dimension, which is characteristic of 3D data visualizations in general as well. We also examine how typical retinal variables affect the legibility of 3D visualizations by investigating visual encodings that map depth information to each of those variables separately and combinatorially. Through such encodings, users are given visual cues to better discern depth locations so that the overall legibility of the visualization can be enhanced.

Specifically, in our 3D legibility framework, we reused the three variables, size, color and (brightness) value, from the 2D legibility framework directly, and added one more variable, transparency, which plays an important role in depth perception in 3D geometries. In our visual encodings, depth information of geometries is encoded either by a single visual variable alone or by multiple variables combined. By comparing different encodings, we examine effects of those visual variables on the depth-separation dimension hence the overall 3D visualization legibility.

3.2. *The Parallel Visualization Pipeline*

Our parallel visualization pipeline is outlined in Figure 1. The parallelism is realized through the MPI infrastructure and the visualization powered by the VTK with

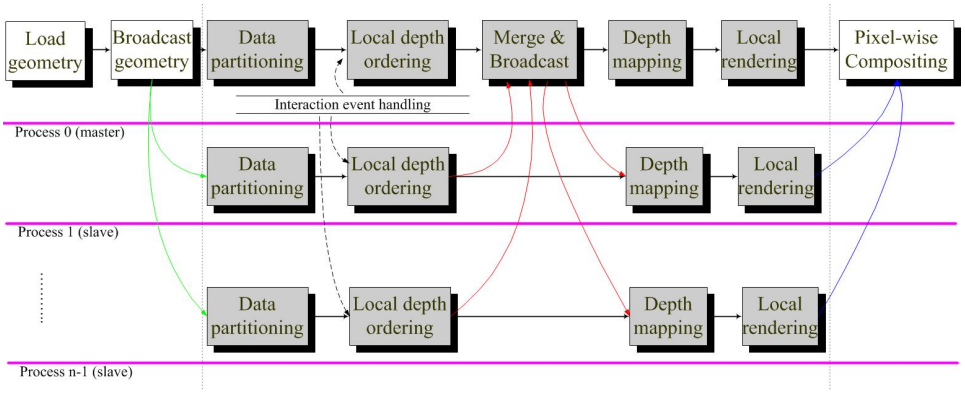

Fig. 1 The overview of our parallel visualization pipeline. The data partitioning is done in the master process: for N processes, the master equally divides the entire data set into N partitions, keeps one partition for itself, and then sends each of the other partitions to one of the $N-1$ slave processes.

parallelization supports. Among the four processes shown in the figure, the master process P_0 is responsible for data I/O, visualization interactions, and coordinations that are required for a parallel rendering with consistent depth mappings, and for rendering local data partitions as also done by all slave processes. The collaborations between the master process and slave processes involve all key steps in the pipeline, from data decomposition to parallel depth sorting and geometry rendering.

3.3. *Data Decomposition*

Data decomposition is usually an essential part of a parallelization mechanism. Although the concrete decomposition scheme can be very much dependent on the interrelations among data components, and there are different levels of granularity at which the data components are defined, it is natural to split the whole data set into independent partitions such that data processing of each partition can be performed in parallel. In the case of 3D geometries, for instance, a single geometry unit (e.g., a polyline or tube in the DTI model) is regarded as a unit component and vertices on a geometry should not be assigned to different partitions.

In addition, to maximally harness the computing resources available, we equally decompose the whole geometry model among all partitions, and then evenly distribute the computational tasks for sorting, mapping, and rendering to all processes. This simple data partitioning scheme is efficient for our case of dense-geometry rendering because, for one thing, there is no data or semantic dependency among all geometries, and for another, task load for each process is closely equal to others even if the master process takes certain additional roles of management and coordination. Moreover, our data decomposition strategy enables the independence of partitions on view updates: each process deals with the partition of the entire geometry model assigned to it, including updating the depth

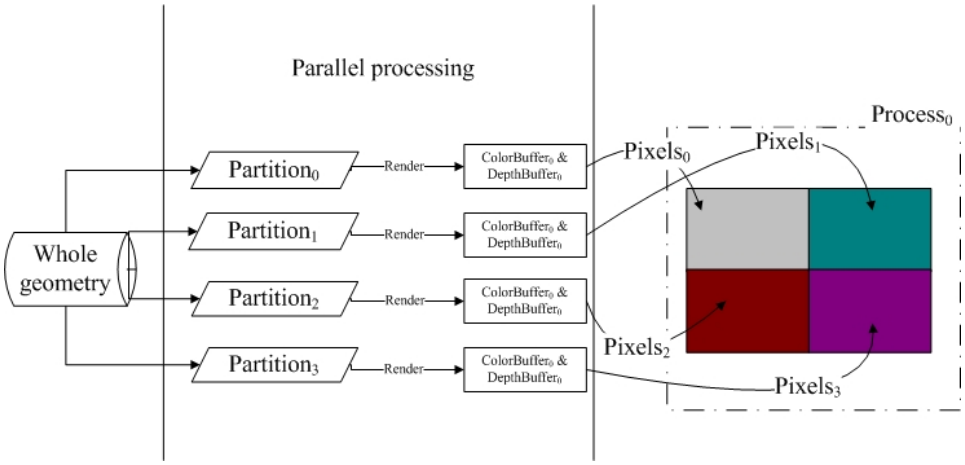

Fig. 2 Illustration of data partitioning and pixel-wise compositing in our framework.

mapping and refreshing the visualization, independently of other processes dealing with the rest of the model; the decomposition scheme itself does not change in the event of viewpoint (camera) changes (arising from user interactions, for instance). However, data decomposition in general itself is a separate topic and there are no universally optimal solutions, and probing the cost-effectiveness tradeoffs and impacts on parallelization performance of different decomposition strategies is out of the scope of this work.

When using MPI as the underlying parallelization support, the visualization data set is decomposed according to the local process id (*LocalProcId*) and total number of processes specified (*ProcNum*). Precisely, suppose a total of n data partitions is intended, given all the data components $C_0, C_1, C_2, \cdots, C_{n-1}$ in the equal-partition scheme, local sub-range of data for process i will be $\{C_{sidx}, C_{eidx}\}$, where $sidx = n/ProcNum * LocalProcId, eidx = n/ProcNum * (LocalProcId + 1)$, for the starting and end index of the data partitions, respectively. Specially, the last process may take more or less data components than others if n is not exactly divided by *ProcNum*, in which case $eidx = n$. Figure 2 illustrates this data decomposition scheme while showing the overall picture about how the depth-stylized visualization is rendered in parallel.

3.4. *Parallel Depth Sorting*

3.4.1. *Per-vertex Depth Ordering*

In application scenarios like the 3D stylized visualization of dense geometries, depth information of each vertex (or other geometry units such as triangles and stripes) can be flexibly mapped to different visual variables, such as size, color, and transparency, to assist with visual perception and navigation. However, despite of which particular mapping is applied, such mappings must be consistent regardless of the current user viewing directions in order to maintain depth perception hence visual legibility along the

depth dimension. In other words, all the vertices (or other geometry units) need to be ordered along the current viewing direction so that they can be consistently mapped to visual variables initially assigned.

Thus, once the viewing direction changes, which typically occurs when users rotates the view, those vertices need to be reordered before mappings are updated to refresh the visualization. Concretely, in the case of dense-geometry rendering, depth ordering for vertices is required if a depth-dependent size encoding is applied. An example of such encodings is that a tube tapers or grows in its radius along the depth dimension.

3.4.2. *Real Time Sorting*

From the above discussion, we can see that, for our visualization scenarios at least, depth ordering is necessary and real-time depth mapping relies on real-time depth sorting. In fact, (re-)sorting attributes (e.g., depth) of geometries (e.g., vertices) is commonly required in interactive 3D visualizations when the attributes change during the interactions to address the object visibility problem. Note that depth buffering (i.e, Z-buffer) alone is not sufficient for our illustrative visualizations since we need to *maintain* consistent visual encodings with respect to various visual elements such as size and color.

For the per-vertex depth sorting, the computation is essentially sorting a sequence of floating-point numbers. For other geometry units, the computation can also be reduced to the problem of per-vertex depth sorting. For instance, if we want to discern the depth locations only at the level of tubes rather than that of vertices (i.e., all vertices on a tube have the same visual-variable value), a vertex can be selected to represent the entire unit, and then the per-geometry sorting becomes sorting the selected representatives—how representative vertices are chosen depends on particular application needs. Therefore, we generalize the depth sorting problem into the sorting of a sequence of cells. Practically, the cell can either contain a single numerical value such as a floating-point number or be a packed data such as a structure including multiple fields. In our case, the concrete structure of the cell is determined by the specific depth mapping or chosen combination of multiple mappings because different mapping requires different granularity at which the visual elements need to be distinguished (e.g., at the levels of vertices, lines, or tubes).

While there are a rich set of parallel sorting algorithms freely available [2], they mostly serve the solitary purpose of sorting and are usually implemented as stand-alone parallel applications. Since our ultimate goal is to parallelize dense geometry visualizations in which depth mappings are integrated, we need a holistic parallel framework where the sorting algorithm works together with other steps, such as depth mapping and parallel rendering, in such a way that maximizes the overall visualization performance. In contrast, we integrated our parallel sorting algorithm into the holistic illustrative visualization pipeline (Section 3.5), that includes data partitioning, and fitted it with efficient depth mappings (Section 3.6.1).

We adopt mixed sorting algorithm for our parallel depth sorting by the following key steps. First, each process updates the depth values (z-coordinates) of local vertices through simple vector arithmetics using the current camera parameters (focal point and

position, etc.). Then, every process sorts vertex depth values in the partition assigned to it using a common `quick-sort` algorithm and then sends the sorted depth information to master process once finished. Finally, the master process gathers locally sorted partitions and performs either a multi-way merge sort or multiple two-way merge sorts. Given our data decomposition strategy, we employed the latter scheme on the master process, which is more efficient because an iterative two-way merging can be performed once a sorted partition is received from a slave process without waiting all processes to finish their local sorting. Algorithm 1 shows how this parallel sorting algorithm works while illustrating how the real-time depth mappings fit the parallel visualization framework as a whole.

3.5. *Parallel Geometry Rendering*

In stylized geometry visualizations, the primary performance challenges come from two sources: depth sorting and geometry rendering. For each updated frame, the whole geometry model needs be rendered over again after depth sorting to reflect the depth mapping updates. Although both are critical for a real-time frame rate, the rendering part (T_r) usually takes a larger proportion of the total frame refreshing time ($T = T_s + T_r$) than the depth sorting time (T_s). In one sample test with a geometry date set of 140,000 vertices, we found that T_s/T was less than 10% in the illustrative visualization where per-vertex depth was mapped to vertex color. This suggests that the rendering phase can be the main bottleneck for the overall visualization performance. In other words, interactive depth-stylized visualization depends on real-time rendering of the depth-mapped geometries. Nevertheless, as mentioned earlier, the proposed approach aims at a cheap and readily applicable parallel illustrative visualization solution based on commodity CPUs, rather than resorting to specialized architecture like GPUs. Our approach to parallel rendering simply consists of two major steps.

3.5.1. *Concurrent Local Rendering*

After data decomposition, we first deploy the local partition to each process. Here we refer to as a process any basic program unit of computation, which can be a single processor on a multiple-processor platform, a single core on a multi-core processor, or a worker thread on a single core processor. For each rendering frame, all separate renditions, each performed by a single process, are aggregated into a single holistic rendering that is only visible on the one of the processes called master process, which is randomly elected from among all processes at runtime. This aggregation is practically realized by means of pixel-wise image compositing as the second step detailed below.

3.5.2. *Pixel-wise Compositing*

When each process finished its local rendering of the partition assigned to it, the rendering ends up with a set of pixels in the frame buffer. As such, pixel-wise compositing is in essence a process of compositing frame buffers. In practice, to reduce computational costs, compositing only the color buffer and depth buffer is sufficient for our visualization

Algorithm 1 integrated parallel depth sorting and mapping

1: $numProcs \leftarrow$ total of processes
2: $myId \leftarrow$ local process rank
3: $numPts \leftarrow$ total number of vertices in local partition
4: Gather all $numPts$ values into array $allNumPts$
5: $idoset \leftarrow 0$
6: **for** $i = 0 \rightarrow myId - 1$ **do**
7: $idoset \leftarrow idoset + allNumPts[i]$
8: **end for**
9: **for** $i = 0 \rightarrow numPts - 1$ **do**
10: $depth[i].vd \leftarrow$ depth value of the ith vertex in local geometries calculated from camera parameters
11: $depth[i].id \leftarrow i + idoset$
12: **end for**
13: sort $depth$ according to the vd field using $qsort$
14: Sum up all $numPts$ to $totalPts$
15: **if** $myId == 0$ **then**
16: $oset \leftarrow 0$
17: $tdepth[0..numPts-1] \leftarrow depth[0..numPts-1]$
18: **for** $i = 1 \rightarrow numProcs - 1$ **do**
19: Receive $tdepth[numPts + oset.numPts + oset + allNumPts[i]]$ from process i
20: inplace merge $tdepth[0..numPts + oset.numPts + oset + allNumPts[i]]$
21: $oset \leftarrow oset + allNumPts[i]$
22: **end for**
23: **for** $i = 0 \rightarrow totalPts - 1$ **do**
24: $hashIndex[tdepth[i].id] \leftarrow i$
25: **end for**
26: Broadcast $hashIndex$
27: **else**
28: Send $depth$ to master process 0
29: Receive $hashIndex$ from master process 0
30: **end if**
31: **for** $i = 0 \rightarrow numPts - 1$ **do**
32: $Rank_{global}[i] \leftarrow hashIndex[i + idoset]$
33: **end for**

purposes. Although in other application contexts alpha buffer may also have to be considered, for brevity, here we describe compositing for these two main types of frame buffer only (alpha buffers can be composited in similar ways).

The compositing process is performed in the following three steps: (1) each process fetches pixels from the frame buffers in its local process memory space; (2) all slave (as opposed to the master) processes send all the buffers to the master process but not its local

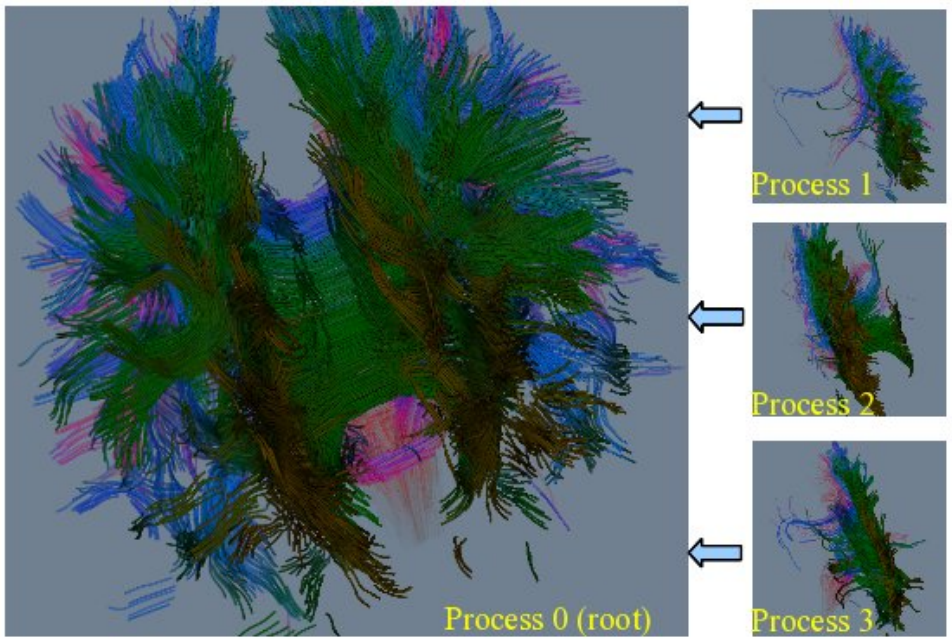

Fig. 3 An example of the pixel-wise compositing in our parallel visualization framework. The dense geometry (tube) visualization with depth mapped to size, color, and transparency (three visual encodings combined) is parallelized using 4 processes. Process 0 (master) gathers all parallel rendering from slave processes and composites them together with its own local rendering to produce the final holistic rendering.

buffers though; and (3) the master process performs a pair-wise buffer compositing every time it receives the buffer from a slave process until all slave buffers are composited. Finally, the master process writes the composited depth and color values back to corresponding local frame buffers to create a complete rendering. Figure 2 illustrates this pixel-wise compositing process, of which an example is shown in Figure 3 where four processes are utilized to parallelize the rendering phase.

In addition, when rendering geometries in parallel in the background, the parallelization should be transparent to users. So except for special needs for showing slave rendering, no rendering partitions should be visible and the composited visualization is displayed on the master process only. Furthermore, there are two optional optimizations for the compositing process, which we have included in our current implementation of the visualization framework. First, off-screen rendering is applied to avoid slave rendering. This is not only to meet the need of slave renderers for invisible rendering but also, and more importantly, to improve the overall rendering performance. Second, creation of rendering windows on all slave processes is avoided. Depending on practical graphics platforms used, a less ideal solution would be to hide the rendering

windows if it is necessary to create them for correct rendering. Example case includes that a window must be created to establish a context for the drawing function to take place in. Finally, synchronizing camera parameters across all processes before any process starts to render can simplify the last step of the compositing. As adopted in our approach, a simple way of realizing this synchronization is to broadcast key camera parameters (focal point and position, for instance) retrieved from the master process to all slave processes.

3.6. *Depth Mappings*

Depth mappings are applied to stylize geometry units according to their depth values so that a better perception in the 3D environment can be obtained. Depending on how the depth value is mapped to the value of different visual variables, a depth mapping is either a linear or non-linear function $f(v) = V(Rank(v_d))$, where $Rank(x)$ is the rank (order) of x in the sorted sequence, v_d is the depth value of a single geometry unit, and function V maps the rank sequence to the domain of the associated visual variable, $[V_{min}, V_{max}]$.[a] In the case of linear mapping, for instance,

$$V(x) = \frac{V_{max} - V_{min}}{x_{max} - x_{min}}(x - x_{min}) \tag{1}$$

When considering size (s), color (c), value (i), and transparency (t) as the visual variables to which depth values are mapped, $V(x)$ is a scalar function. Also, $V(x)$ is a unitary function for single mapping, while multiple mappings are simply an aggregation of multiple single mappings. For example, when mapping depth to size, color, and transparency at the same time, $V : x \to (s,c,t)$ is simply $V : x \to (S(x), C(x), T(x))$, where S, C, T are all unitary mappings for the size, color, and transparency, respectively.

3.6.1. *Depth Mappings in Parallel Rendering*

In the context of geometry rendering, depth mappings are easily performed according to the simple function evaluations as described above. However, depth mappings need be parallelized as well in order to collaborate with parallel rendering for optimized performance of the illustrative visualization. In our parallel visualization framework, depth mappings are required to be coherent in the geometry model as a whole. Therefore, simply mapping local geometry on each process independently and then compositing the locally depth-stylized rendering would not produce correct visualizations.

On each individual process, the input of depth mapping is the rank of depth values of local geometries and, as a result, each process will only have the local rank for every vertex in its local geometry partition. However, the global rank of a vertex in the range of the whole geometry must be retrieved for a coherent global depth mapping. With global ranks of local vertices, every process can render its local geometry independently yet correctly due to the correct mappings from the local vertices to the partition of the range

[a]For brevity, we mostly use the case of vertex to illustrate our approach, which however can be applied similarly to other types of geometry units.

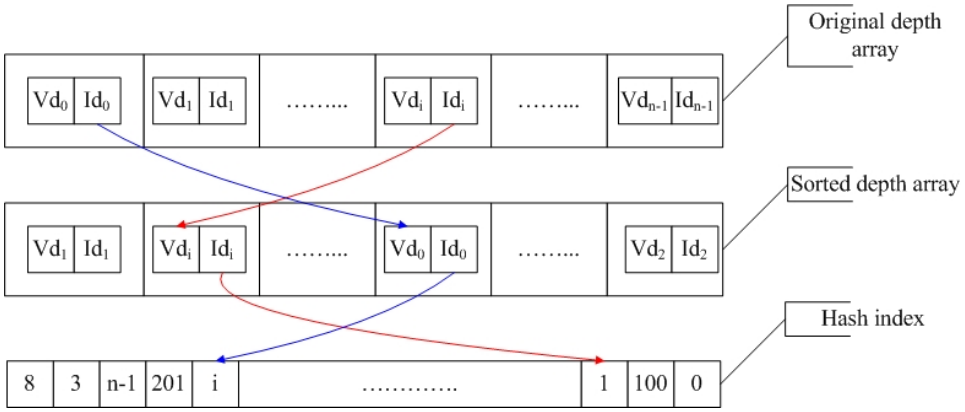

Fig. 4 Hash index for efficient depth mapping in support of interactive illustrative visualizations in our framework.

of $V(Rank(v_d))$ corresponding to those vertices. Figure 4 shows the outline of the integrated parallel sorting and depth mapping algorithm in our parallel visualization.

3.6.2. *Hash Index*

To obtain the global depth-sorting rank of each local vertex, a straightforward solution would be to gather all locally sorted partitions together onto the master process, sort all depth values, and then broadcast the resulting ranking, stored in an array, to all other processes. As such, the global rank would be retrieved from the depth ranking received for evaluating $f(v)$ for each vertex v. However, the retrieval would be of a $O(N^2)$ cost for all N local vertices, which can be too prohibitive to reach an interactive frame refreshing. Instead, for a real-time global depth ranking, we create a global hash index for the whole geometry immediately after the depth sorting on the master process completed.

In both the local and global depth arrays, an index is kept for each depth value at each element and the depth array is essentially a sequence of vector (d, Id) where d is the depth value and Id is the index, which is initialized with the global depth rank of a vertex in the original input geometry model. As such, wherever a depth array element is moved after sorting, its original rank, taken as a vertex identifier as well, can be always retrieved immediately. We use this id to associate the unsorted with sorted depth array through the hash index. Figure 4 illustrates this hashing process used for our depth mappings.

3.6.3. *Depth-mapping Updating*

As we discussed earlier, during the interactive exploration of the depth-stylized visualizations, mappings need be updated whenever the depth order of geometries along the current viewing direction changes, and the mapping updating is then reflected through rendering refreshing. For that purpose, our technique actively triggers the frame

refreshing once the mapping is updated. Two strategies are available for the synchronization between frame refreshing and mapping updating such that the former is immediately triggered upon the occurrence of the latter.

First, a polygonal data filter, which is used for depth sorting, can be inserted into the demand-driven rendering pipeline so that rendering update will be triggered when either the input or output of the filter is modified. However, besides updating depth mappings, geometry cloning between the data filters is also required, which can greatly slow down the visualization frame rate. It is noteworthy here that the geometry itself is not updated at all when the depth mappings change. The other mechanism is to explicitly invoke frame refreshing via user-interaction handling, with which only mappings are recomputed while no geometry cloning is involved. We employ the latter for a better performance. In addition, for the interaction-driven updating strategy, we only directly handle user inputs, such as mouse interaction, that may change the depth order of geometries on the master process. When responding to such user inputs, the master process invokes frame updating after finishing mapping calculations and then sends a remote method invocation (RMI) message to all slave processes. In the RMI handler on each process, mapping updating is first triggered, followed by an active call to frame updating.

4. Implementation

Our parallel depth-stylized visualization is implemented in C/C++ using VTK with parallelization supports of MPI. In the parallel sorting algorithm, the *qsort* routine is adopted from the standard C library for local `quick-sort` on each process, and generic in-place `merge` algorithm in C++ STL library is used for iterative two-way merge sort on the master process. We have employed image-compositing functionalities provided by VTK with necessary extensions that tailor their functions for our customized pipeline components in order to implement the pixel-wise compositing.

In addition, our depth sorting filter is extended from VTK's filter for polygonal data depth sorting and also from an interactor component extended from VTK's track-ball camera interactor, which work together to realize the interaction-driven mapping updating mechanism. To explicitly trigger frame updating, user-defined RMI messages are added and the callbacks are registered to VTK's multiple process controller before parallel rendering starts. With these extended components, the interactor responds to data rotation by broadcasting a mapping-updating RMI message to all slave processes, and then mapping calculations and frame update are invoked in the callback of the RMI message. The visualization program is simply running as a MPI application, thus the number of processes can be specified when launching the MPI runtime. As we discuss in detail in Section 5, an optimal number of processes to be used depends on the configurations of underlying hardware architecture.

Figure 5 shows the outlook of our parallel visualization interface (as in other sample visualizations, we omit the colormap to highlight the rendering itself; our system provides options to show or hide colormaps). The GUI is created using Qt 4.0 by which all the interaction widgets are set up for the depth-stylizing customizations. In order to achieve

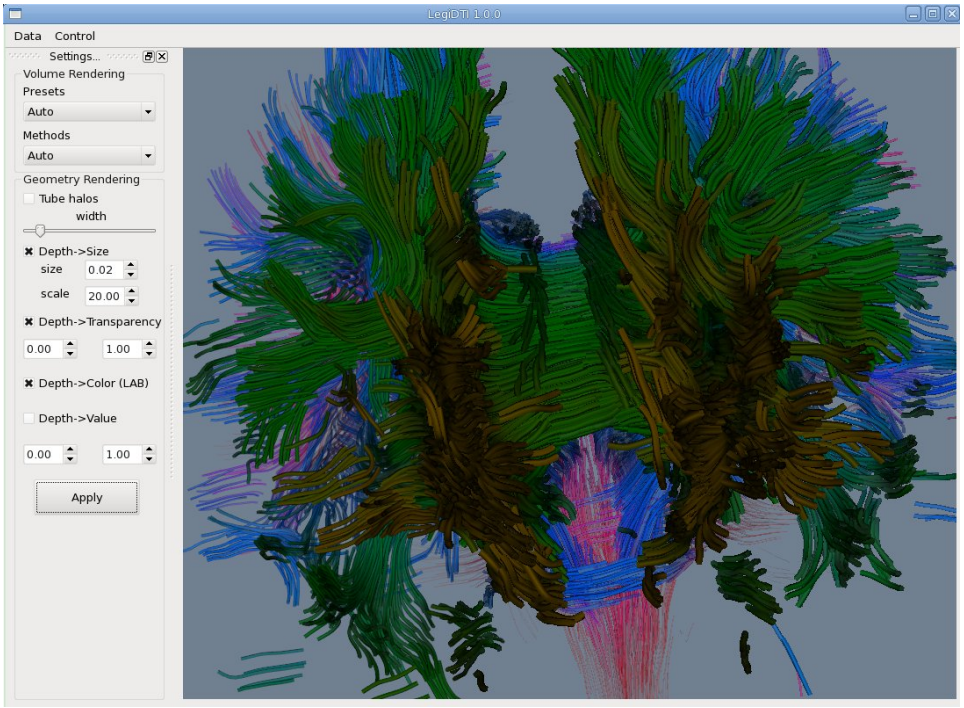

Fig. 5 The outlook of our parallel depth-stylized 3D geometry visualization interface.

better performance, parallel processing is applied only to the rendering widget and all other GUI components are created on the master process only. GUI interactions have to be explicitly relayed from the master process where they are triggered to all slave processes so that the slave renderings can reflect changes in the illustrative rendering settings as a result of those interactions. To that end, we register another type of RMI message and define a callback devoted to realizing the RMI for updating slave renderings. RMI messages are easily transmitted by MPI communications. Note that beyond the visual encodings and illustrative geometry rendering we presented in this paper, our framework has also integrated additional features, such as tube halos and volume rendering. We omit discussion on them in this paper in order to focus on the central topic.

5. Empirical Results

We have applied our parallel visualization framework to interactive depth-stylized 3D tube visualizations of DTI data sets with single and multiple depth encoding schemes applied in order to enhance users' depth perception in the 3D visualizations. Figure 6 shows sample renderings produced by our framework for one of the DTI data sets.

Additionally, using the implementation as described above, we evaluated the efficiency of our parallel visualization approach by first measuring the overall rendering

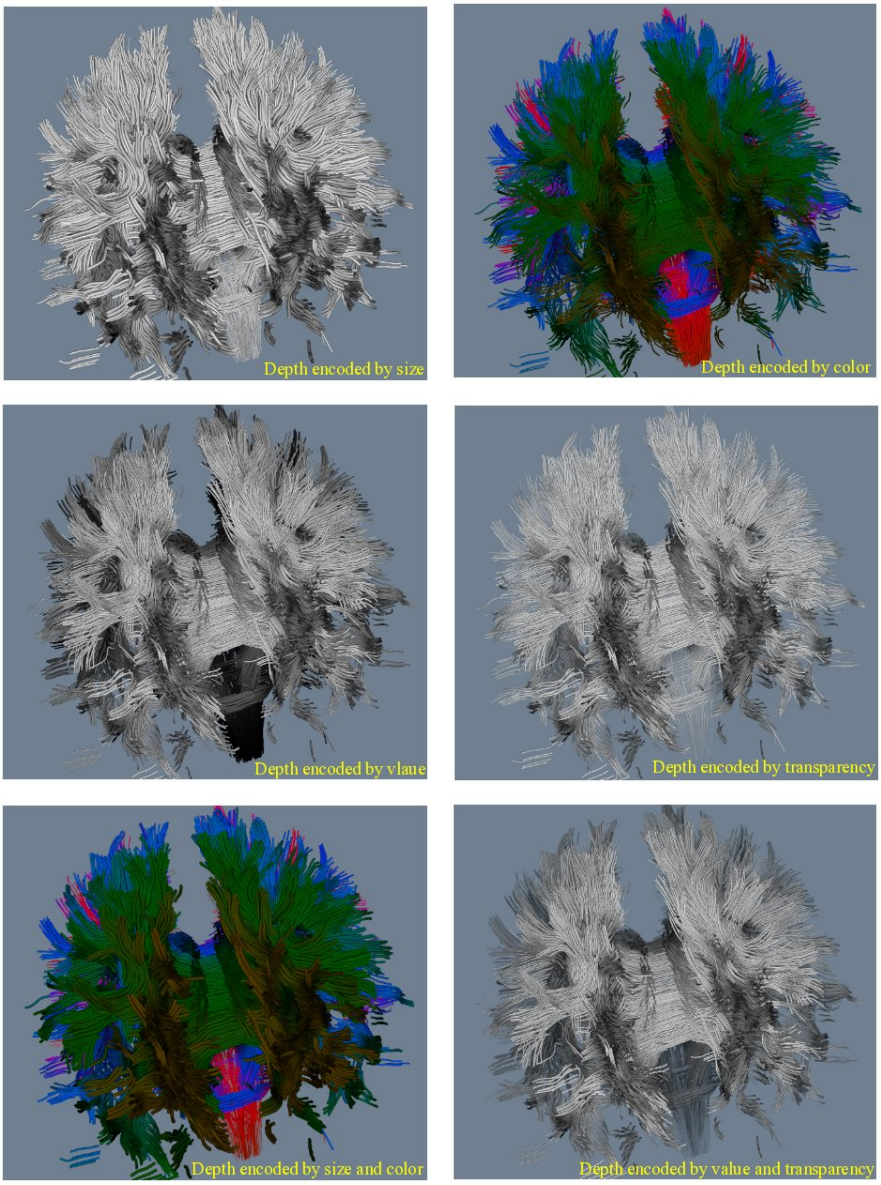

Fig. 6 Our parallel visualization of a DTI model using tube shape-encoding with single mapping, including depth to size (upper left), color (upper right), value (middle left), and transparency (middle right), respectively, and multiple mappings, including depth to size and color combined (bottom left) and to value and transparency combined (bottom right). We use these different mappings with typical visual variables to communicate depth information in this 3D DTI tube-based visualizations.

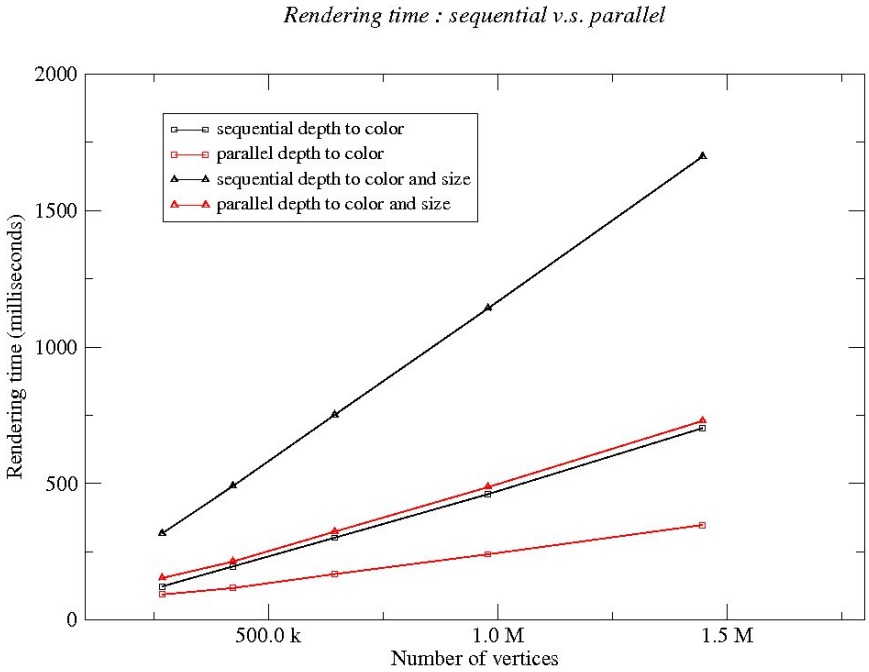

Fig. 7 Performance for depth-stylized geometry visualizations using our parallel method compared with sequential visualization approach. Both single depth encoding and multiple depth encodings are applied in the experiment and performance results compared between the two (parallel and sequential) visualization approaches.

cost, including that of depth sorting and MPI communication, and then comparing our method to other alternative parallel rendering approaches. Next, we report the results based on many runs of illustrative rendering of a selected DTI model on an Intel(R) Core(TM)2 Quad 2.66GHz processor with 4GB DDR2 memory.

5.1. *Parallel Performance*

We measured the proposed parallel visualization method by first comparing its rendering performance against sequential alternatives, for different scales of 3D geometry data sets. Specifically, for each test data set, the total time (in milliseconds) spent by our approach for rendering a single frame of the interactive visualization is compared to that by the sequential alternative for performing the same computation. Here in our application scenarios, we visualize 3D depth-stylized tubes generated from diffusion tensor MRI data with different depth mapping schemes applied for the tube illustrations.

As shown in Figure 7, parallelization enables interactive rendering performance for our depth-stylized geometry visualizations, which is hard to obtain with a sequential approach. Each number of the rendering-time measures is an average of the total rendering cost over

Table 1 Effects of the number of processes on the parallel performance in terms of time costs in milliseconds, on parallel speedup, and on efficiency. The data points were obtained from an experiment running our parallel visualization framework for the visualizations of 9,635 tubes consisting of 1,447,005 vertices, with depth mapped to color.

Metrics	Number of Processes					
	2	3	4	5	8	12
Time (ms)	409	359	347	401	469	642
Speedup	1.72	1.95	2.02	1.75	1.5	1.09
Efficiency	0.86	0.65	0.51	0.35	0.19	0.09

100 continuous frames. For the parallel rendering, the time measure included the cost of communications among the (four) processes we used in our experiments.

Without loss of generality, we differentiate only two instances of visual encoding schemes here, depth to color only and depth to size and color combined. These two instances may represent two disparate types of computations in terms of complexity for depth encodings in our study. For the single-variable mapping, from depth to color, there is only one pass of depth sorting beyond the rendering phase. For the multiple-variable mappings, from depth to both size and color at the same time, there are two passes of depth sorting plus the tube mesh generation besides the rendering phase. Of these two passes of depth sorting, one is for mapping the depth of geometries to the sizes of them before geometries of different radii are generated. The other is for mapping the depth of tube geometries to the color of them after tube geometries are generated from polylines (the input format of the geometry model).

To examine the effects of the number of processes on the visualization performance, we run the comparative study with different numbers of processes used in our parallel rendering pipeline. The main results are shown in Table 1. As can be seen, performance increases monotonically with the number of processes before that number reaches four. However, beyond four processes, the performance decreases, also monotonically, when the number continues to grow. This may be explained by the fact that the CPU we used has four cores—running the visualizations on a CPU of more cores is expected to see more speedups before reaching the cutoff point. On the other than, it seems to be counterintuitive that with four cores the parallelization achieved a speedup of 2x only (while ideally it should have been around 4x). The main reason for the unideal speedup is that, according to the architecture of our framework, the rendering of all data partitions is not fully parallel—the pixel compositing task needs to wait for the other rendering tasks to complete before it can start, and the compositing task itself, which is more expensive than the slave rendering tasks, is not parallelized.

It is also important to note that although we used a quad-core CPU for this experiment, and, as the results suggest, the architecture (e.g., number of cores) of computing hardware does affect the visualization performance, our technique itself does not require the CPU to have multiple cores, nor does it have any other specific

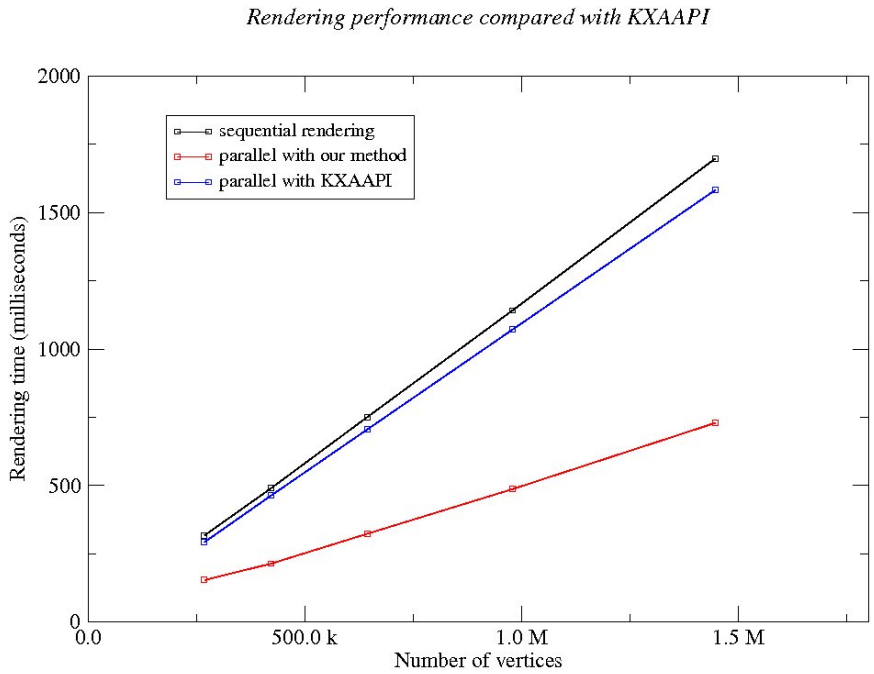

Fig. 8 Performance for partially-parallel depth-stylized geometry visualizations using our approach compared to that using KXAAPI with respect to the same computations (for depth sorting). Performance of the sequential approach is also included for reference.

requirements concerning the hardware architecture—the goal of our approach is to offer an efficient illustrative rendering solution for legible visualizations in dense geometry environments by maximally harnessing any commodity CPUs available to users.

5.2. *Peer Comparisons*

Beyond the performance comparisons between our technique and *sequential* alternatives, we further investigated the efficiency of the proposed approach to peer *parallel* alternatives, for depth-stylized geometry rendering. We implemented the 3D geometry visualization with depth-stylizing using both partially- and fully-parallel rendering. For both configurations, we gauged the total rendering time with five different scales of 3D tube geometries stylized by depth-dependent color and/or size encodings, similar to the methodology for measuring performance gain of parallel over sequential visualizations presented in Section 5.1. We also used the same experiment settings as those in the first study, including the quad-core CPU.

For the partially-parallel configuration, only the depth sorting is parallelized while the rendering phase is sequential, with which we intend to show the advantages of our approach with respect to meshing the sorting parallelization with the rendering parallelization. We employed the Kernel for Adaptive, Asynchronous Parallel and

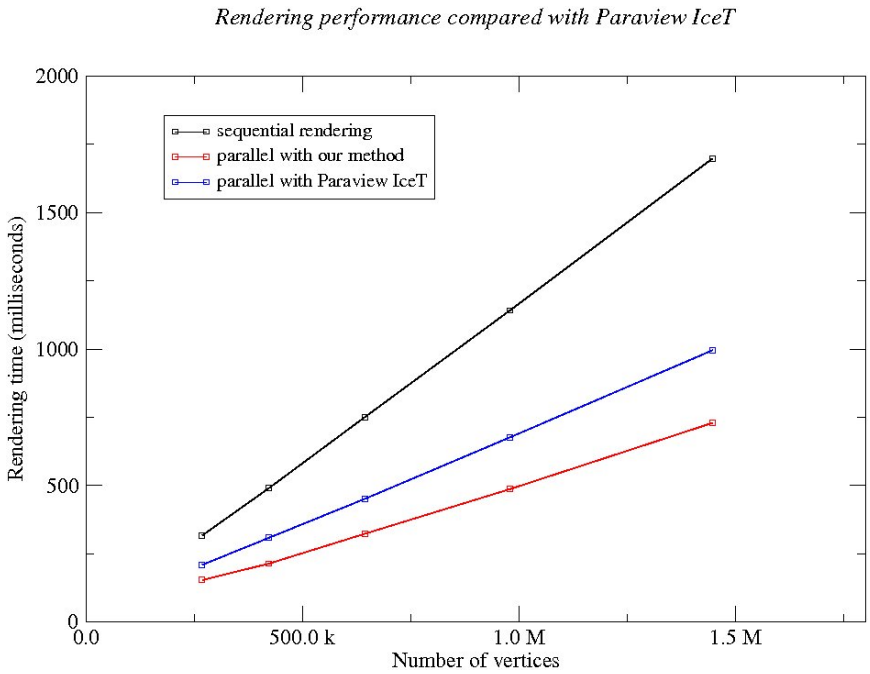

Fig. 9 Performance of fully-parallel depth-stylized geometry visualization using our approach compared to that of Paraview IceT with respect to the same computational tasks. Performance of the sequential approach is also included for reference.

Interactive programming (KXAAPI) framework [11] to sort the depth information of the whole geometry model on the sequential visualization pipeline of VTK. The major result is depicted in Figure 8, where the y axis indicates the execution time taken by the three alternative sorting approaches while the x axis lists different scales of geometry in terms of the numbers of vertices sorted. As can be seen, the KXAAPI took almost as much the time for the sorting as the sequential algorithm, and the growth of time expense with the growing geometry sizes is also very close to the sequential approach. A possible reason is that the KXAAPI framework focuses on the overall parallel programming facilities rather than sorting performance only. In contrast, the results suggest that our approach appears to be much more efficient, outperforming the peer solution by over two times: not only did it take significantly less time on any of these five data sets, the time cost also grows much slower when the data size increase, suggesting a better scalability of our algorithm.

For the fully-parallel configuration, both depth sorting and the overall rendering pipeline are parallelized. For an alternative fully-parallel visualization solution to compare, we implemented our depth-stylized geometry visualization using the IceT module in Paraview [17]. We used the same performance metrics used in comparative study with the parallel sorting solution KXAAPI. Specifically, we ported part of the IceT module as necessary from the Paraview source package to VTK for our experimentation.

Figure 9 shows the comparisons in the same format as Figure 8. The results clearly exhibit the advantages of parallelization over sequential computation—there is a large gap between these two types of computing schemes. Nevertheless, our parallel visualization solution largely outperformed the IceT based parallelization constantly. In addition, our approach also appears more scalable than the peer solution, although not as much as seen in the comparisons to partially-parallel approaches.

6. Discussion

In our application scenarios, we visualize depth-stylized 3D geometries, which is generated on the fly during visualization rendering. Alternatively, tube meshes can be produced beforehand to avoid the computational cost of tube generation during the rendering process. However, we did not adopt off-line tube generation due to performance considerations. During the interactive visualization, we need to change visual variable values (e.g., geometry sizes like tube radii) upon depth changes for depth encodings, for which manipulating geometries during the rendering process can help obtain higher overall visualization performance than loading preprocessed geometries. For example, it is more efficient to load line geometries and then generate tubes on the fly for visualizing geometries with depth to tube size mapping than loading tube meshes as original inputs and then transforming each geometry to implement the depth encoding.

One interesting observation from our experiment results is that, as the geometry model size scales up, the visualization performance accelerations are much greater when multiple-variable depth mappings are applied than when the depth is mapped to one visual variable at a time only. A possible explanation is that larger portions of the overall computation tend to be parallelized with multiple-variable mappings than with single-variable ones, hence the larger gain in the overall rendering performance. This may suggest that it is promising to scale our framework to more sophisticated visualization contexts where more computation-intensive steps associated with rendering are involved.

Based on the scalability results shown in Table 1, one may extrapolate that the visualization performance may continue to grow beyond four processes if we increase the number of cores in the CPU used in the experiment. Also, in terms of implementation, given the scalability of the underlying MPI infrastructure to more advanced computing architectures, including multiple processors, the performance could be further accelerated on those architectures. However, as we noted before, we do not intend to develop a parallel visualization solution superior to solutions leveraging sophisticated hardware such as GPUs, but rather to offer a solution as efficient as possible on cheap commodity architectures. In this sense, using our approach on an advanced computing platform may not be the best option: a GPU-based solution may give better performance. On the other hand, users may consider adopting our framework based on their budgets. For example, using advanced CPUs (such as those of many-cores) to obtain better performance with our framework may not be worth it: GPU-based solutions can be more cost-effective if those advanced CPUs are even more expensive than GPUs.

7. Related Work

7.1. *Depth Enhancement*

There has been much previous work focused on technical solutions to the depth perception issues and visual occlusions in 3D data visualizations. For instance, a rich set of landmarks and context cues [20] and shading with transparency [12] both help enhance depth perception while alleviating occlusion problems within overlapping structures. Focusing on improving depth perception also, Bruckner et al. employ volumetric halos to increase the 3D legibility of volume visualizations [3]. They introduce different halos according to different ways of halo-volume combination, and use halos to construct inconsistent lighting, which accentuates depth even further from a different perspective.

Elmqvist et al. [10] give a thorough discussion about occlusion management in 3D visualizations, where they focused on reducing 3D occlusions. Occlusion management for visualization is a more general class of visibility problem in computer graphics, which is concerned about improving human perception for specialized visual tasks, such as occlusion, size and shape. This method extensively enhanced the legibility of 3D data visualizations. In contrast, we investigate how to manipulate typical retinal variables [13] in graphics perception to address depth-wise legibility.

As visualization data sets usually include a large number of overlapping structures, even direct volume rendering techniques can also suffer from poor depth cues. Using MIP (maximum intensity projection) rendering [14], however, only few effort is required to create a good understanding of the structures represented by high signal intensities. This algorithm adds two different visual cues, occlusion revealing and depth based color. For the first cue, the MIP color of occluding objects is modified for the same materials as those at points of maximum intensity; for the second, actual positions of shaded fragments are used to change their color using a spherical map. In this paper, we explore depth enhancement in dense geometry visualizations by encoding depth information using various visual variables.

Ritter et al. [26] employ hatching strokes to communicate shape while using distance-encoded shadow to further enhance depth perception in their vascular-structure visualizations. In addition, they achieve real-time rendering performance using GPU-based hatching algorithm, which is efficient for rendering complex tabular structures with depth being emphasized. Similarly, we handle tabular shapes in our visualization scenarios also, but intend to improve depth perception in a much denser 3D (geometry) date set derived from human brain MRI models. More importantly, our approach provides a cheaper interactive rendering solution that works on commodity CPUs in comparison to the GPU rendering facilities they employed.

7.2. *Parallel Visualization*

Parallelization has been extensively harnessed in data visualizations, especially where performance becomes a challenge. In [1], the authors developed a scalable and portable parallel visualization system based on augmenting VTK for efficiently visualizing large

scale time-varying data. The system they proposed provides parallelism on both task and pipeline levels, which primarily addresses visualization programmers. Also at a system scale but even earlier, SCIRun [15] had offered task and data parallelism as a data-flow-based visualization system running on shared-memory machines with multiprocessors. This system was later extended to support task parallelism on distributed-memory architectures [21]. In comparison, we present a lightweight parallelization method for visualizations of large 3D geometries by using existing facilities, such as MPI and VTK, instead of providing a full-flown integrated system or extended programming library.

Compared to system-level solutions, a lot more parallelization efforts for visualization focus on parallel rendering, ranging from photo-realistic rendering [25] to volume rendering [29] and parallel iso-surfacing [23]. Other researchers have probed in more indirect approaches, such as image composition schemes [18] and data decomposition strategies [28], to improve polygon rendering performance. More recently, various parallel rendering algorithms, including sort-first, sort-last, and the synthesis of them, were used and evaluated on shared-memory computers, yet these algorithms originally targeted distributed-memory architectures [22]. In our work, we also explore polygon rendering parallelization and employ image composition, but we aim particularly at depth enhancement for the purpose of providing more legible 3D geometry visualizations, by overlapping parallel depth sorting with parallel polygonal data rendering.

Note that the parallel sorting problem [19,9], which is at the core of our parallelization framework here, could be alternatively solved with possibly high efficiency on GPU platforms using a rich set of existing algorithms [24,27]. In this paper, we target a cheaper solution without relying on high-end computing resources such as GPUs. Alternatively and complementarily, we use CPU-based parallel sorting algorithms leveraging a single processor of either single or multiple cores, which is widely available in almost any modern computers with bottom-line hardware configurations.

8. Conclusion and Future Work

We presented a parallel illustrative visualization framework that enables interactive frame rates in legibly rendering dense 3D geometries. The framework leverages commodity computing architectures to offer rendering efficiency for interactive visualizations, and utilizes flexible depth encodings to enhance the visual legibility in the dense 3D environment. Our approach has been evaluated using dense geometry models containing millions of vertices with multiple mappings from geometry depth information to various visual variables, and shown to be effective for addressing performance issues often seen in legible dense 3D visualizations like the depth-stylized illustrations we explored in this paper. We also demonstrated the advantages of our approach, as a commodity-CPU-based parallel visualization framework, over both sequential alternatives and peer parallelization approaches including XKAAPI and Paraview Icet. Our results suggest that the proposed framework can provide an effective option for parallel visualizations, especially when only cheap commodity computing hardware is available.

The presented evaluation has been focused on the rendering performance of the proposed parallelization approach, with the empirical assessment on the improvement in visual perception supposedly provided by the illustrative visualizations is left for future work. An immediate next step would be to conduct an user study that consists of two components: a qualitative survey collecting subjective user opinions on the usefulness of various legibility enhancements of our visualizations, and a quantitative experiment measuring how the enhanced visual perception improves users' understanding of the visualized data set, with respect to a set of predefined user tasks [8], such as identifying an anomaly in the brain MRI model and recognizing the difference in seeding resolution applied during the MRI data acquisition [6]. More specifically, for the quantitative study, the task performance can be gauged in terms of the accuracy and time cost for each task and compared between two groups where one group uses the enhanced features under test while the other is given the visualizations with corresponding features disabled.

Acknowledgments

This manuscript has been considerably improved thanks to the valuable suggestions and insightful comments given by anonymous reviewers.

References

1. J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A parallel approach for efficiently visualizing extremely large, time-varying datasets. *Los Alamos National Laboratory, Los Alamos, New Mexico, Technical Report LAUR-00-1620*, 2000.
2. G. E. Blelloch. A sampling of parallel sorting algorithms. `http://www.cs.cmu.edu/~scandal/nesl/algorithms.html#sort`, 2014.
3. S. Bruckner and E. Gröller. Enhancing depth-perception with flexible volumetric halos. *IEEE Transactions on Visualization and Computer Graphics*, pages 1344–1351, 2007.
4. H. Cai. Zifazah: A scientific visualization language for tensor field visualizations. Master's thesis, University of Southern Mississippi, 2012.
5. H. Cai, J. Chen, A. P. Auchus, S. Correia, and D. H. Laidlaw. InBox: In-situ multiple-selection and multiple-view exploration of diffusion tensor MRI visualization. In *IEEE Symposium on Biological Data Visualization*, 2011.
6. H. Cai, J. Chen, A. P. Auchus, J. Huang, and D. H. Laidlaw. Measuring seeding resolution dependence of diffusion tensor streamtube visualization. In *IEEE Visualization Poster Compendium*, 2011.
7. H. Cai, J. Chen, A. P. Auchus, and D. H. Laidlaw. InShape: In-situ shape-based interactive multiple-view exploration of diffusion MRI visualizations. In *International Symposium on Visual Computing*, pages 706–715, 7 2012.
8. J. Chen, H. Cai, A. P. Auchus, and D. H. Laidlaw. Effects of stereo and screen size on the legibility of three-dimensional streamtube visualization. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2130–2139, 2012.
9. E. K. Donald. The art of computer programming. *Sorting and searching*, 3:426–458, 1999.
10. N. Elmqvist and P. Tsigas. A taxonomy of 3d occlusion management for visualization. *IEEE Transactions on Visualization and Computer Graphics*, pages 1095–1109, 2008.
11. T. Gautier. Kernel for adaptative, asynchronous parallel and interactive programming. `http://kaapi.gforge.inria.fr/dokuwiki/doku.php?id=start`, 2014.

12. P. Irani and C. Iturriaga. Labeling nodes in 3d diagrams: Using transparency for text legibility and node visibility. Technical report, University of New Brunswick, 2002.
13. B. Jacques. *Semiology of graphics: diagrams, networks, maps*. University of Wisconsin Press, 1983.
14. J.Diaz and P. vazquez. Depth-enhanced maximum intensity projection. *International Symposium on Volume Graphics*, 8:1–8, 2010.
15. C. Johnson and S. Parker. The SCIRun parallel scientific computing problem solving environment. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
16. Kitware. The visualization toolkit. `http://www.kitware.com/opensource/vtk.html`, 2014.
17. Kitware. VTK/MultiPass rendering with IceT. `http://www.vtk.org/Wiki/VTK/MultiPass_Rendering_With_IceT`, 2014.
18. T. Lee, C. Raghavendra, and J. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):202–217, 1996.
19. C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to algorithms*. The MIT press, 2001.
20. Y. Li, C. Fu, and A. Hanson. Scalable wim: Effective exploration in large-scale astrophysical environments. *IEEE Transactions on Visualization and Computer Graphics*, pages 1005–1012, 2006.
21. M. Miller, C. Hansen, and C. Johnson. Simulation steering with scirun in a distributed environment. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 366–376, 1998.
22. B. Nouanesengsy, J. P. Ahrens, J. Woodring, and H.-W. Shen. Revisiting parallel rendering for shared memory machines. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 31–40, 2011.
23. S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of IEEE Visualization*, pages 233–238. IEEE, 1998.
24. M. Pharr and R. Fernando. *GPU Gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.
25. E. Reinhard, A. Chalmers, and F. Jansen. Overview of parallel photo-realistic graphics. *Eurographics 98 State of the Art Reports*, pages 1–25, 1998.
26. F. Ritter, C. Hansen, V. Dicken, O. Konrad, B. Preim, and H. Peitgen. Real-time illustration of vascular structures. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):877–884, 2006.
27. E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.
28. S. Whitman. Dynamic load balancing for parallel polygon rendering. *IEEE Computer Graphics and Applications*, 14(4):41–48, 1994.
29. C. Wittenbrink. Survey of parallel volume rendering algorithms. Technical report, Hewlett-Packard Laboratories, 1998.