Eurographics Symposium on Parallel Graphics and Visualization (2006), pp. 1–8
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

Dynamic load balancing for parallel volume rendering

Paper 1022

Abstract

Parallel volume rendering is one of the most efficient techniques to achieve real time visualization of large datasets by distributing the data and the rendering process over a cluster of machines. However, when using level of detail techniques or when zooming on parts of the datasets, load unbalance becomes a challenging issue that has not been widely studied in the context of hardware-based rendering. In this paper, we address this issue and show how to achieve good load balancing for parallel level of detail volume rendering. We do so by dynamically distributing the data among the rendering nodes according to the load of the previous frame. We illustrate the efficiency of our technique on large datasets.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation – Viewing algorithms I.3.2 [Computer Graphics]: Graphics Systems – Distributed/network graphics

1. Introduction

With the advent of high performance interconnection networks in recent years, clusters have become an inexpensive alternative to supercomputers. More recently, improvements in consumer graphics hardware allow the use of clusters as a cost effective solution for real-time visualization by adding a consumer-grade graphics card to each node.

In the field of visualization, parallel rendering allows interactive visualization of large datasets at a high quality over a cluster of workstations, which could not be done on a single machine. Techniques for doing parallel rendering are usually classified into three groups according to the place of the sorting phase in the graphics pipeline as done by Molnar *et al* [MCEF94]. If sorting is done prior to transforming and rasterizing the primitives, the approach is of the sort-first kind. If sorting is done between the transformation and rasterization phases, it is of the sort-middle kind. If sorting is done after rasterizing the primitives, the approach is called sort-last. However, only sort-first and sort-last techniques make sense for volume rendering. In this paper, we focus on sort-last techniques. We motivate this choice in section 2.

In the field of volume rendering, level of detail techniques are widely used to visualize large datasets on a single machine [WWH*00, PTCF02, LHJ99] by decomposing these datasets into bricks, and by using level of detail techniques. However, using such techniques together with distributed rendering results in serious load unbalance. For instance, when a static data distribution is chosen, if the user zooms on parts of the model, one node might spend significantly more rendering time than the others because its visible data share is bigger, while other nodes could be idling because their data share is not visible. Similarly, if a level of detail approach is used, a single node might have to render its data share at a higher resolution than the other nodes. Obviously, in these two situations, load imbalance slows down the whole rendering process and is not desirable. For this reason, level of detail techniques have rarely been used in the context of parallel volume rendering, while they could be extremely valuable to achieve visualization of very large scale datasets. In this paper, we overcome this issue by proposing a technique to compute an approximate load balancing for sort-last parallel volume rendering. This technique uses a time coherent dynamic data distribution to achieve good load balance.

The paper is organized as follows : related works are introduced in section 2. Section 3 describes our load balancing algorithm in detail. Section 4 is dedicated to implementation details and experimental results. We have experimented with 1GB datasets on a cluster with up to 16 nodes. Finally, concluding remarks and open issues are discussed in section 5.

2. Related works and motivation

In the field of sequential volume rendering, numerous techniques allow visualizing large datasets on a single machine using simplification-based methods. Weiler *et al* [WWH*00] and Lamar *et al* [LHJ99] use different resolution textures

depending on factors like the distance to the observer or the brick contents, thereby reducing the total requirements for texture memory. This allows better frame rates and larger datasets visualization. Guthe *et al* [GS04] use advanced techniques such as occlusion culling and empty space skipping to further speed up rendering. Strengert *et al* [SMW*04] propose an efficient hierarchical sort-last volume rendering technique, and report interactive results on a Myrinet interconnection network. Lamar *et al* [LHJ03] propose an efficient error computation technique for 3D data. This technique first computes an histogram over each brick, and subsequently uses the fact that evaluating the error over an histogram of the values in a given brick is faster than evaluating the error at each voxel. In particular, this technique can be used to quickly find the parts of the data that are not visible with respect to the current transfer function by simply applying this transfer function to the histogram. However, using such techniques as-is in a parallel visualization environment results in a serious load imbalance between the computation nodes.

Parallel rendering techniques are usually classified into three groups according to the classification done by Molnar *et al* in [MCEF94] : sort-first, sort-middle and sort-last rendering techniques. However, only sort-last and sort-first apply in the parallel volume rendering context.

- In the *sort-first* situation, primitives are distributed among the nodes at the beginning of the rendering pipeline, usually by splitting the screen into regions and associating each region to one node. In this approach, load balancing can be achieved by dynamically splitting the screen into rectangular regions as done by Samanta *et al* [SZF*99]. However, such a dynamic splitting does not withdraw the main drawback of the sort-first approach, i.e. lots of data redistribution happens as shown by Bethel *et al* [BHPB03].
- In the *sort-last* situation, the data is split between the nodes, and each node renders its own portion. Then, compositing takes the depth information into account to form a final image from each node's rendering. Sort-last volume rendering techniques are able to handle very large datasets as demonstrated by Wylie *et al* [WPLM01] by statically distributing these datasets among the nodes. Compression has also been used to push the data size limit further by Strengert *et al* [SMW*04]. However, such sort-last techniques do not have a good load balance between the nodes. Hence when only some parts of the data are visible, or when some parts of the data are rendered using a lower level of detail then others, serious load imbalance can occur. Wang *et al* [WGS04] achieve dynamic load balancing in the context of software based volume rendering using a space-filling curve. However, such an approach cannot be used directly on graphics hardware, since it would result in lots of pixel readbacks. Lee *et al* [LSH05] achieve static load balancing of volume rendering by hierarchically subdividing the data.

Our objective is to propose a parallel visualization technique for volume rendering which guarantees load balancing without inducing too many data communications that would penalize the whole rendering process. Since sort-first volume rendering imposes large data redistribution, we will focus on sort-last techniques which do not require such data redistribution.

The main issue is therefore to present a technique to achieve load balancing in a sort-last context. The following phases have to be taken into account in parallel rendering : rasterization (including frame buffer readback), communication and compositing. When focusing on large dataset visualization, the frame time is dominated by the rendering phases. This is the reason why we concentrate on the rasterization phase. Notice moreover that load balancing of the communications and compositing phases has already been achieved by Stompel *et al* in [SLM*03]. To do so, the authors split the rendered data into pixel spans, and then compute a schedule of the compositing of these spans minimizing the imbalance between the nodes.

Level-of-detail techniques are crucial to efficient visualization of large datasets. However, when combining level-of-detail techniques and parallelism, the rendering time over the nodes may vary widely : a node in charge of a higher detailed area will take significantly more time to render its data than a node in charge of a lower detailed area. In order to avoid this load unbalance, we propose a technique which consists in dynamically redistributing the data among the nodes, so as to guarantee load balance. The contribution of this paper is therefore to propose a load-balanced, out-of-core, parallel level-of-detail technique. We also measure the optimal brick size performance-wise for gigabyte-sized datasets.

3. Load balanced parallel rendering algorithm

Our algorithm overview is as follows : before rendering, the data is split into bricks of equal size. During rendering, before each frame, an approximation of the rendering cost is used to build the load balanced data distribution. Rendering is then performed and compositing is finally achieved in a sort-last fashion. In the next two subsections we detail the data distribution algorithm and then describe how load balancing is achieved using a client-server approach. In particular, the evaluation of the rendering cost is of uttermost importance to achieve good load balancing.

3.1. Data distribution and caching

The data is partitioned into equally-sized bricks of voxels, and bricks that cross the dataset borders are padded with empty voxels. These bricks are used as the basic data element in the whole distributed graphics pipeline, from data access up to the graphics hardware's 3D textures. All these bricks have the same dimensions, which have to be a power

of two. This constraint is inherited from the graphics hardware which imposes power of two for 3D texture sizes (as it ls the case on the GeForce FX cards we use). Notice however that our method can handle bricks of arbitrary sizes. In order to be able to compute gradients on the boundaries, these bricks overlap by one voxel. The granularity we subsequently use for data distribution is one brick. Using a brick granularity, we are thereby able to discard bricks that are not visible with respect to the current transfer function or the current viewing conditions. For example, if a brick is fully transparent it can be discarded. To achieve this, we have implemented the technique described by Lamar *et al* in [LHJ03]. This technique associates an histogram of the data with each brick. This histogram can then be convoluted with the current transfer function to determine the brick's visibility. Similarly, bricks that fall outside the view frustum can be discarded.

The bricks are dynamically distributed among the clients. To achieve high performance in data distribution, we use a multi-layered cache for the data bricks. This approach is depicted in figure 1 : the data bricks are initially replicated from the file server to each client's hard disk, which makes subsequent access to the data significantly faster than fetching it through the network. Then at run time, bricks are fetched out-of-core from disk and are cached at two different levels : in system RAM, and in video RAM. The bricks are kept in video memory following a LRU (least recently used) policy.

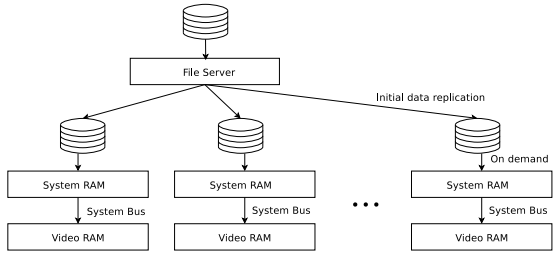

Figure 1: *Hierarchical cache layout*

3.2. Load balancing

To achieve load balancing, we need to evaluate the rendering cost, that is to quantify the workload for rendering specific parts of the dataset. Finding such a quantification function is very complex. A series of benchmarks have been conducted in order to find the influence of the visualization parameters (texture size, screen size, texture contents, viewing angle, visibility of the data...) on the rendering time. Some of the results are summarized in figure 2, and show that the rendering time cannot be easily predicted. In particular, the viewing angle can affect rendering time by a factor of more than 3. Similarly, the viewing distance is not easily correlated to the rendering time for a given brick. Therefore, it is impractical to predict an accurate cost function using the visualization parameters. Thus, instead of trying to predict the workload, we use the rendering time of the previous frame as an estimation of the cost, and use it to adjust the load balancing for the next frame.

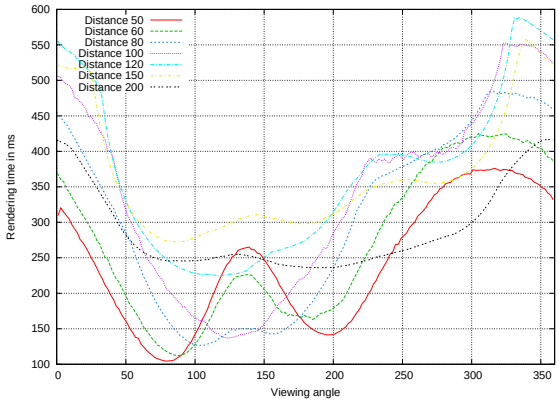

Figure 2: *The rendering time for a brick, depending on two viewing parameters (angle and distance)*

Our load balancing technique is based on a kd-tree decomposition of the data space. A kd-tree is a binary tree where at each level the data is split along a plane which is orthogonal to one of the base axes (*Ox*,*Oy*,*Oz*). The data is split along each of these axes in an alternating fashion. If the depth of the tree is larger than 3, the splitting planes are used circularly. This ensures temporal locality of the decomposition when the tree is rebalanced. Changing the plane direction at the same level of the tree between two subsequent frames would result in a lot of data redistribution, and thus is avoided.

The algorithm is implemented in a client-server fashion as depicted on figure 3. The server's role is to build the kd-tree while the clients are in charge of rendering and compositing. Each client holds a portion of the data which is a parallelepiped set of bricks that we call a *zone*. Initially, since there is no information on the rendering times, the server setups the kd-tree so that it decomposes the data into equally-sized *zones*. Then, after each rendered image, the rendering and readback times are communicated back from the clients to the server, which uses them to re-balance the kd-tree. Once the tree has been entirely traversed, the server sends the data distribution to each of the clients by sending the extremal points defining the *zone*. Each client loads the appropriate bricks, proceeds to render its *zone* and finally performs the compositing on the video buffers using the binary-swap sort-last algorithm proposed by Ma *et al* [MPHK94]. Since the *zones* generated by the Kd-tree are all parallelepipeds, and thus are all convex, compositing is trivially achieved by sorting the buffers generated by the clients according to their distance to the observer. The resulting frame is then sent to the server for final display.

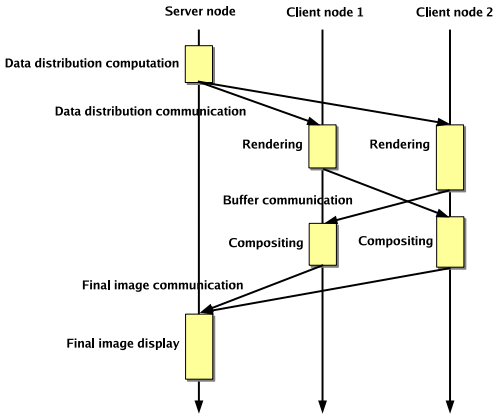

Figure 3: *Overview of the load balancing technique over the course of one frame*

Let us now detail how the costs are used on the kd-tree to rebalance the load. The rendering time information is added to each kd-tree node : each leaf node is associated with a client and holds the actual cost for its *zone*, and each internal tree node holds the sum of the costs for its sub-tree. We use an additive metric to compute the cost of an internal tree node because rendering costs are additive. That is, we consider that a given workload can be spread on a number of nodes, and the sum of the computation times on these nodes is the same as the original workload computation time. Indeed, if a client were to render all the bricks associated with a subtree, the rendering time would be the sum of the rendering times of the tree leaves. This assertion has been observed experimentally for hardware-based graphics rendering, as long as the volume of data can be held in graphics memory. The purpose of our algorithm is therefore to have balanced costs on all the leaf nodes, thereby achieving good load balance.

Initially, since there is no rendering cost information, the server sets the same cost for all the nodes, as shown on figure 4 for eight clients. Then, after the initial frame is rendered and for all subsequent frames, the client nodes communicate back the rendering costs to the server, which places them on the leaf nodes of the kd-tree. The costs are then propagated by adding them upwards the tree as shown on figure 4 and the server uses these costs to rebalance the kd-tree. The tree is parsed using a depth-first traversal during which each internal node is examined and balanced according to the cost of its children in the following way : the rendering cost for the *zones* of the two children are compared and the separating plane between these two *zones* is moved by one slice of bricks in order to reduce the cost of the most expensive *zone* and correlatively increase the cost of the cheapest one. The plane is only moved by one slice at a time to avoid causing too much data loading for each frame and also to preserve temporal data coherency. Thereby, the algorithm initially converges towards a balanced state in a small number of frames, and subsequent adjustments are small enough to avoid disturbing real time rendering.

Figure 5 illustrates the technique. This figure shows two subsequent frames and the corresponding brick decomposition (in gray), kd-trees and resulting data distributions. In a depth-first traversal, let us first consider node I and its two children. Since the cost for A is bigger than the cost for B, the kd-tree splitting plane between *zones* A and B (depicted in green on figure 5) is moved by one slice of bricks towards A to reduce the workload for A and increase the workload for B. This shift is realized along the plane orthogonal to *Ox*. In the same way, when considering node J and its two children, because node D is more expensive than node C, the plane is moved towards D (as shown in dark blue on figure 5). Moving up the tree and considering node M, the algorithm checks the costs of nodes I and J, and since the cost of J is larger than that of I, the plane orthogonal to *Oz* (depicted in yellow on the figure) is moved towards I. Node N is treated the same way. Then at the next step when moving up to node O, the plane orthogonal to *Oy* (shown in orange on the figure) is moved toward N since the cost of N is larger than that of M. The resulting data distribution is used to compute the next frame. The corresponding renderings are shown on figure 6, with each node drawing in a different color in order to highlight the respective *zones*.

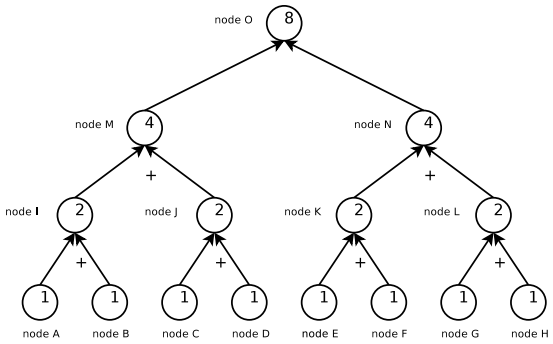

Figure 4: *Propagating the load values by adding them upwards the tree*

4. Implementation and results

4.1. Implementation

We have implemented our algorithm in C++. The communication layer was written using the socket API, and we use OpenGL for rendering. Since the rendering phase mainly takes place on the GPU and the communication phase is handled by the CPU, it makes sense to overlap them. To achieve good overlapping between rendering and communication, each node runs three threads. The first thread handles

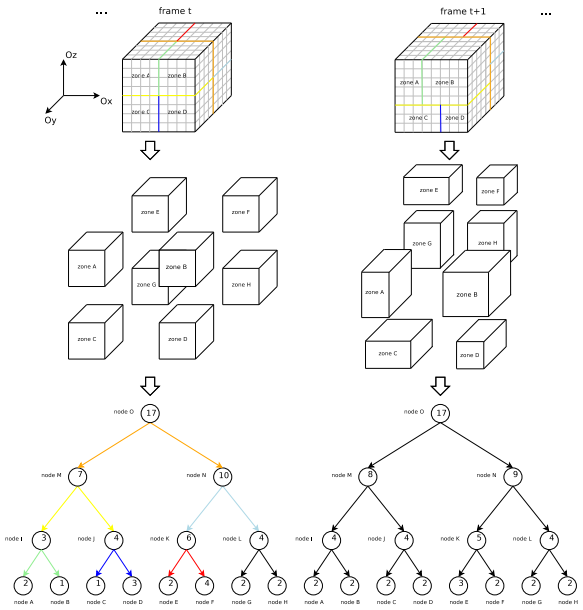

Figure 5: *Load balancing the kd-tree according to the cost function during two subsequent frames*

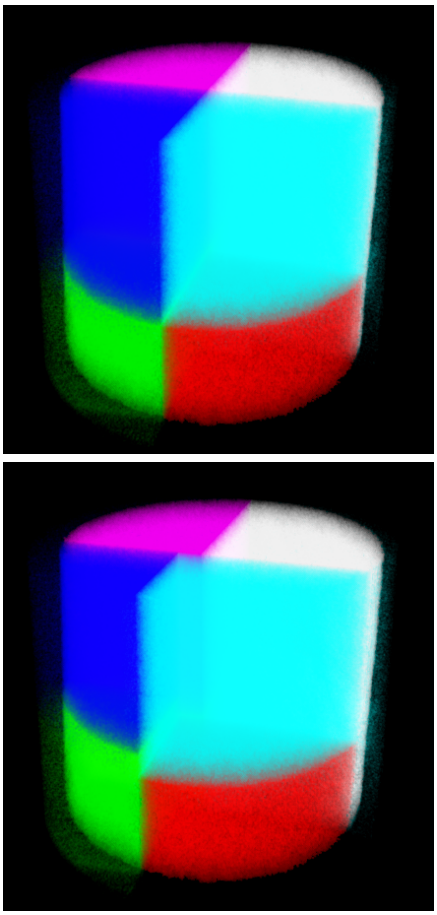

Figure 6: *Resulting rendering from the previous load balancing operation. Top : unbalanced. Bottom : balanced. Each client is configured to draw in a different color*

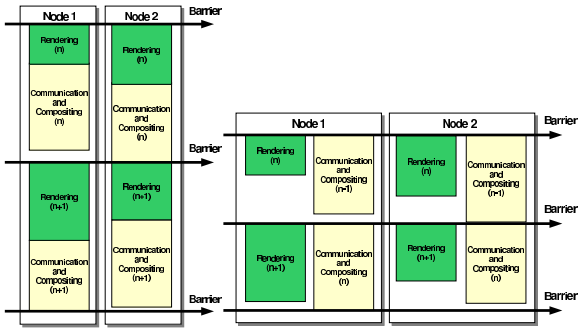

Figure 7: *Classical (left) vs threaded (right) approach*

rendering ; the second thread handles compositing and communication and the third thread handles data loading. As figure 7 shows, using threads for the rendering and communication processes allows efficient overlapping of the communication phase with the rendering phase. The high-latency network-bound communication phase mainly uses the network card, while the GPU-bound rendering phase uses the GPU exclusively for rendering. Thus, we overlap these two phases by creating one thread for each of them, which results in a speedup even on a single CPU machine. This is especially worthy since both communication and rendering can be blocking operations that would otherwise slow down rendering. Another thread is created which asynchronously fetches the data. The compositing algorithm used is the binary-swap [MPHK94] technique with bounding box optimization.

To achieve out-of-core data loading and memory caching, we use the mmap() system call to access to the datasets on each of the nodes. We have found mmap() to be significantly faster than accessing the data file randomly using the fopen() and fseek() calls.

4.2. Results

We have tested our algorithm over a cluster of machines running Linux connected by a gigabit ethernet network. The hardware configuration details are described in table 1. To improve the gigabit ethernet network performance, we have enabled jumbo frames on the switch and on the machines used for the test, and we have increased the size of the send and receive queues for the network interfaces to 2000, as well as the size of the kernel network memory buffers to 1 megabyte, and we have disabled the selective acknowl-

edge (sack) algorithm. To achieve lower latency, we have disabled the nagle algorithm by setting the TCP_NODELAY socket option. All tests were conducted using a 1 gigabyte ($1024 \times 1024 \times 1024$ voxels) geological dataset obtained from X-Ray imaging. This dataset depicts a geological core. All renderings were done into a 1024*768 viewport.

Component	Type
CPU	1*Athlon XP 3000+
Memory	1GB
Network	Gigabit Ethernet
Network card	Intel pro 1000 MT
Graphics card	GeForce FX 5900 XT
Graphics memory	128 MB

Table 1: *Hardware configuration of a cluster node*

Figure 8 demonstrates the scalability of our implementation by showing the average rendering times over a precomputed path for 2, 4, 8 and 16 nodes.

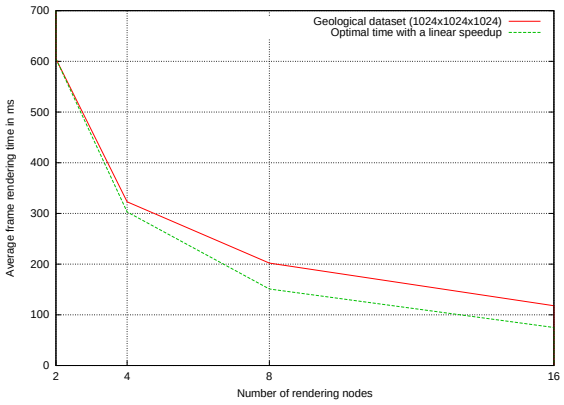

Figure 8: *Rendering times for the 1GB geological dataset*

Figure 9 compares threaded vs unthreaded versions of the code. Using a threaded approach results in a speedup between approximately 5% and 10%. It is interesting to notice the influence of threading with respect to the network jittering. With the non threaded approach, peaks in the graph are only going upwards, while in the threaded approach, a peak upwards is followed by a peak downwards. This is because the threaded approach is able to compute the next frame during the high network latency period, and thus is able to send the frame over the network as soon as the network is ready, thus resulting in a downward spike.

We have tested different brick sizes ($32 \times 32 \times 32$, $64 \times 32 \times 32$, $64 \times 64 \times 32$, $64 \times 64 \times 64$ and $128 \times 64 \times 64$) in order to find the right balance between small bricks (which allow finer-grained load balancing but are more costly since there is a per-brick overhead) and large bricks (which have less overhead but have a coarser load balancing granularity).

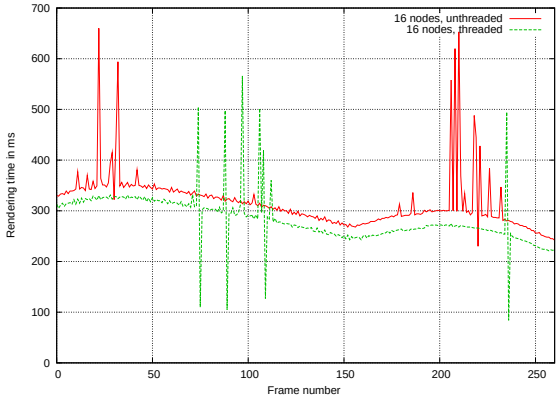

Figure 9: *Threaded vs. unthreaded performance*

Figure 10 shows these measurements taken over a precomputed path which starts far away from the data viewing the full set, and zooms on small parts. These results show that a brick size of $64 \times 64 \times 64$ is a good choice since it results in the best performance.

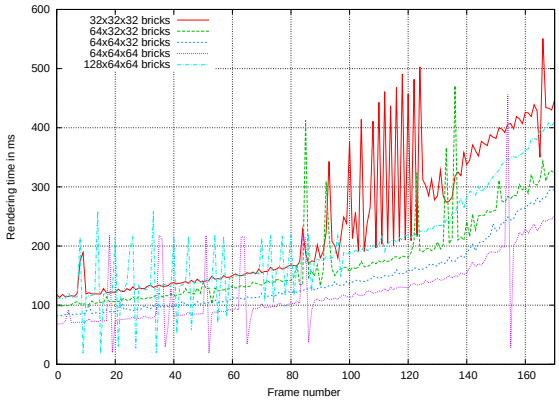

Figure 10: *The influence of the brick size*

To show the influence of load balancing in the context of parallel volume rendering, we ran the same camera path that zooms on the model, with and without our load balancing technique, and using 8 and 16 nodes. As the observer gets closer to the model, the bigger the load imbalance is, and the more relevant a load balancing techniques becomes. These results are shown on figure 11. The balanced algorithm shows good performance since in the best case it reduces the frame rendering time by a factor roughly equal to 4. Moreover, it shows that 8 processors with load balancing outperform 16 processors without load balancing. This is due to the choice of the path which zooms on parts of the data : as the observer gets closer to the object, only a fraction of the data remains visible. Thus, in an unbalanced approach, this results in most of the clients having almost nothing to

render due to the invisibility of their *zone*. Obviously, only a few processors are then in charge of most of the actual rendering which results in slowdowns. On the various curves random peaks occur at some points. This is mainly due to network jittering. Such peaks, when their magnitude is large enough, are noticeable by the user, and could probably be removed using a dedicated interconnection network. Another remarkable result of figure 11 is that for the balanced algorithm 16 nodes outperform 8 nodes by a factor bigger than 2. We attribute this to the fact that the 1GB dataset does not fit within the 8 card's video memory, and thus cause texture trashing. On the other hand with 16 nodes the dataset fits into the video cards memory completely and thus no texture uploads have to take place. Figure 12 presents a per-node breakup of the rendering time, showing that in the non-balanced case, a single node (node 5) is in charge of most of the rendering work, while in the balanced case, the workload has been spread among all the nodes, thus resulting in a speedup.

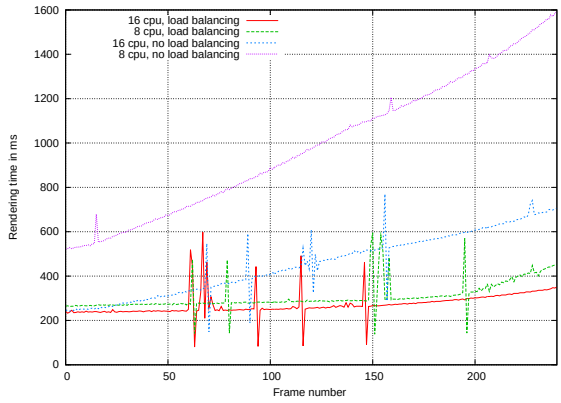

Figure 11: *The influence of load balancing when zooming on part of the data*

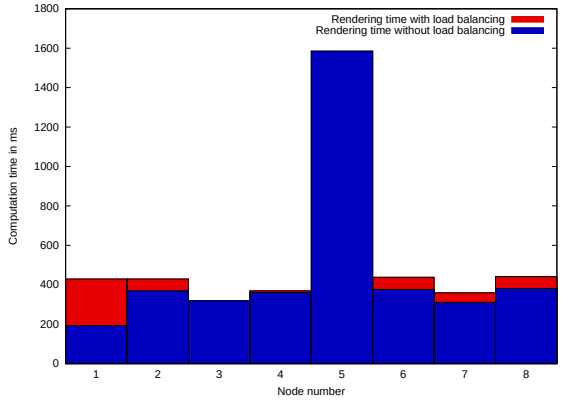

Figure 12: *Balanced vs unbalanced breakup of the rendering times*

Finally, thanks to our approach, we were able to render the full dataset at interactive frame rates while keeping the high detail of geological structures, as shown on figure 13. Interactive frame rates of approximately 5 frames per second are achieved using preintegrated rendering and shading when viewing the gigabyte dataset. Such frame rates are obtained even during close-up examinations, as opposed to approximately 1 frame per second with the non balanced approach. Figure 14 shows the load balanced decomposition obtained using the $256 \times 256 \times 256$ bonsai dataset.

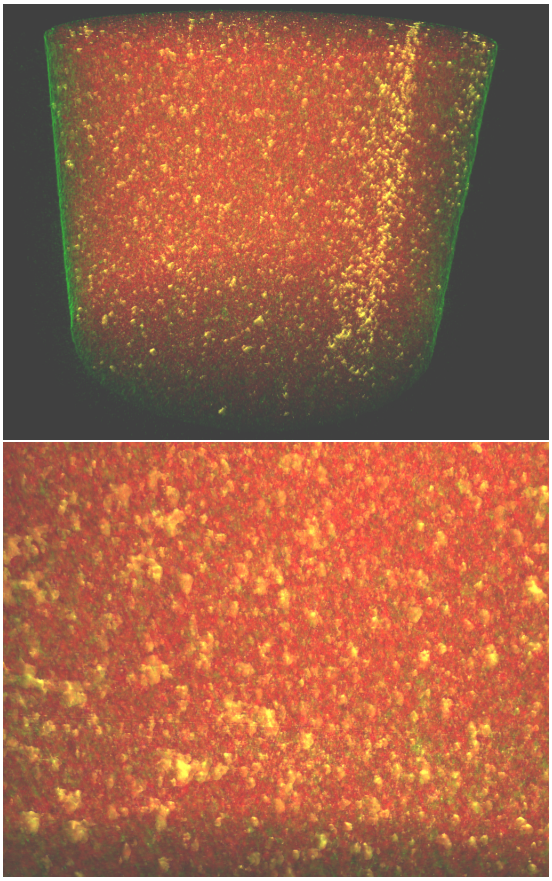

Figure 13: *Rendering examples for the geological dataset. Top : full dataset. Bottom : close-up on small structures*

5. Conclusions and future works

We have presented a method for load balancing parallel volume rendering which ensures good load balance when used together with level of detail or when viewing only parts of a large dataset. This method relies on two points which are tightly coupled : a load balancing technique and a data caching and prediction technique based on a kd-tree decomposition. We also manage to totally avoid any preprocessing phase which could be prohibitive given the size of

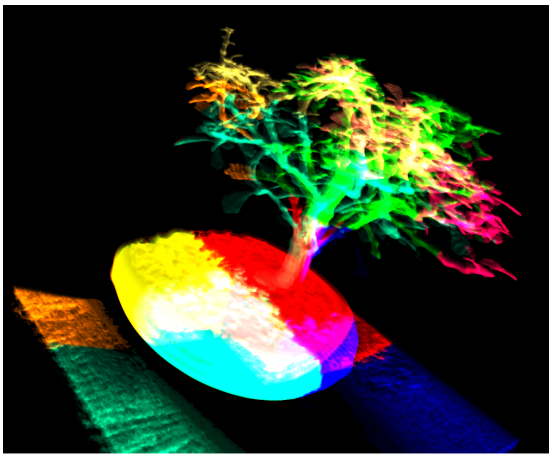

Figure 14: *Load balanced decomposition of the bonsai dataset.*

the datasets. This method proves particularly efficient when zooming on large datasets, or when viewing parts of out-of-core datasets.

Further improvements to our technique are possible. Thanks to our out-of-core data caching and prefetching system, our approach would suit very well to temporal datasets. Also, since it automatically adapts the workload to each nodes computing power, we would like to experiment our load balancing technique on heterogeneous clusters. In particular, we think our technique could handle a network of heterogeneous machines without needing to explicitly measure the respective performance of the nodes, but by adapting the respective computation loads for each node on the fly.

References

[BHPB03] BETHEL E. W., HUMPHREYS G., PAUL B. E., BREDERSON J. D.: Sort-first, distributed memory parallel visualization and rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 41–50.

[GS04] GUTHE S., STRASSER W.: Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics 28*, 1 (Feb. 2004), 51–58.

[LHJ99] LAMAR E., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of the IEEE Visualization conference* (1999), D. Ebert M. G., Hamann B., (Eds.), pp. 355–362.

[LHJ03] LAMAR E. C., HAMANN B., JOY K. I.: *Efficient Error Calculation for Multiresolution Texture-Based Volume Visualization*. Springer-Verlag, Heidelberg, Germany, 2003, pp. 51–62.

[LSH05] LEE W.-J., SRINI V. P., HAN T.-D.: Adaptive and scalable load balancing scheme for sort-last parallel volume rendering on gpu clusters, 2005.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 23–32.

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 59–68.

[PTCF02] PLATE J., TIRTASANA M., CARMONA R., FRÖHLICH B.: Octreemizer: a hierarchical approach for interactive roaming through very large volumes. In *Proceedings of the symposium on Data Visualisation 2002* (2002), Eurographics Association, pp. 53–ff.

[SLM*03] STOMPEL A., LUM E., MA K.-L., AHRENS J., PATCHETT J.: SLIC: Scheduled linear image compositing for parallel vollume rendering. Parallel Visualization and Graphics 2003, IEEE.

[SMW*04] STRENGERT M., MAGALLÓN M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical visualization and compression of large volume datasets using gpu clusters. In *EGPGV* (2004), pp. 41–48.

[SZF*99] SAMANTA R., ZHENG J., FUNKHOUSER T., LI K., SINGH J. P.: Load balancing for multi-projector rendering systems. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1999), ACM Press, pp. 107–116.

[WGS04] WANG C., GAO J., SHEN H.-W.: Parallel multiresolution volume rendering of large data sets with error-guided load balancing. In *EGPGV* (2004), pp. 23–30.

[WPLM01] WYLIE B., PAVLAKOS C., LEWIS V., MORELAND K.: Scalable rendering on pc clusters. *IEEE Comput. Graph. Appl. 21*, 4 (2001), 62–70.

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMANN K., ERTL T.: Level-of-detail volume rendering via 3d textures. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization* (New York, NY, USA, 2000), ACM Press, pp. 7–13.