Eurographics Symposium on Parallel Graphics and Visualization (2007)
Jean M. Favre, Luis Paulo dos Santos, and Dirk Reiners (Editors)

Hybrid CPU-GPU Unstructured Meshes Parallel Volume Rendering on PC Clusters

M. Juliachs[1], T. Carrard[2] and J.-P. Nominé[2]

[1]Laboratoire PRiSM, Université de Versailles, France
[2]CEA/DIF, Bruyères-le-Châtel, France

Abstract

Large-scale numerical simulation produces datasets with ever-growing size and complexity. In particular, unstructured meshes are encountered in many applications. Volume rendering provides a way to efficiently analyze such datasets. Recent advances in graphics hardware have enabled the implementation of efficient unstructured volume rendering algorithms on the GPU. However, GPU architecture limitations make these methods difficultly amenable to a parallel implementation, which is necessary to render very large datasets at interactive speeds and high resolutions. Many previous parallel approaches have focused on software-based algorithms. In this paper, we present a hybrid object-space/image-space CPU-GPU distributed parallel volume rendering method, taking advantage of the flexibility afforded by the CPU, including SIMD processing capabilities, and using GPUs to perform repetitive tasks like depth-sorting and compositing. We present the impact of the different phases on the overall rendering time as a function of node number.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Distributed/network graphics, I.3.8 [Computer Graphics]: Applications

1. Introduction

Numerical simulation of unsteady physical phenomena routinely produces datasets with ever-increasing element number and complexity. Datasets based on unstructured meshes are encountered in many application domains. We want to focus on volume rendering for such datasets with a large number of elements, ranging from 10^6 to 10^8 elements, and a high-dynamic range, both in time and space. During the last few years, the dramatic increase in graphics processors (GPU) computing speed and features has enabled the development of novel unstructured volume rendering algorithms, taking advantage of GPU user-programmable features to reach interactive rendering speeds for relatively modest-sized tetrahedral datasets. Unfortunately, their performance is often not high enough yet to be able to render the largest datasets generated by large-scale numerical simulation at interactive rates (at least 1 frame rendered/s). By distributing the rendering workload and dataset, parallel rendering allows to aggregate the rendering performance of N nodes.

Several uniform grids parallel volume rendering algorithms have been developed on PC clusters with GPUs. These methods generally partition the whole dataset into sub-volumes distributed among the N nodes. As in sequential rendering, the global visibility ordering can be determined from the position of each sub-volume in the global volume relative to the viewpoint. Enforcing a global visibility ordering is more difficult with unstructured volume rendering, as mesh geometric and topological characteristics such as non-convex boundaries, holes, or multiple disconnected components require suited visibility ordering algorithms. Moreover, parallel rendering dataset partitioning may generate further irregularities such as sawtooth boundaries. For these reasons, parallel unstructured volume rendering is difficult to perform on GPU clusters, as sequential algorithms generally execute all of the rendering pipeline on the GPU and do not easily allow retrieving intermediate data which might be required by a parallel implementation. A hybrid object/image-space parallel algorithm retains advantages from both object-space and image-space based rendering algorithms:

- it allows an arbitrary partition of the dataset between projection nodes, thus allowing dataset size to scale with the number of nodes,
- it determines a per-pixel image-space depth ordering

of tetrahedron ray-segment-equivalent contributions: this ordering is unaffected by mesh geometric and topological particularities and dataset partitioning.

We base our parallel rendering approach on an existing distributed rendering algorithm [MC97], which uses a hybrid object-space/image-space 2-stage pipeline. Our contribution is two-fold: first, we use the SIMD instructions available in modern CPUs [Int05] to speed up the tetrahedron projection object-space stage. Then, we execute the image-space stage on the GPU, including ray segment sorting and compositing.

In this paper, we first present related work, including relevant single-GPU algorithms and existing parallel unstructured volume rendering algorithms. In the following section, we present our CPU/GPU object-space/image-space hybrid approach. We then present performance results, including scalability tests as a function of node number. Finally, we discuss the limitations of our implementation, and how to overcome them, as well as how our approach could be extended to take advantage of current and upcoming architectures.

2. Related work

Most of the research work in parallel volume rendering on distributed architectures has been done on rendering algorithms for uniform grids, using a sort-last rendering architecture [SMW*04]. This architecture consists in redistributing the pixels of partial images, rendered at the end of the graphics pipeline, in order to generate a final image by compositing [MCE*94]. Usually, the whole dataset is partitioned into a set of sub-volumes between N nodes, each generating a partial image. The depth-ordering of all of the sub-volumes with respect to the viewpoint gives the correct composition order of the N partial images.

Most existing parallel volume rendering algorithms for unstructured meshes have been developed on specialized architectures (either shared- or distributed-memory). An image-space raycasting algorithm was developed by Ma [Ma95] on a distributed memory architecture, partitioning the dataset and the image-space between N nodes. Each node performs raycasting on its local dataset subset, producing ray segments, and also composites all the ray segments lists associated to the pixels belonging to its image part. As a given ray segment may belong to a pixel composited by another node, a N-to-N ray-segment redistribution phase occurs between raycasting and compositing. Furthermore, the two stages are overlapped in order to decrease rendering time. Building upon this approach, Ma et al. [MC97] have developed a hybrid objet-space/image-space algorithm on a distributed memory architecture. It arbitrarily partitions the dataset between N nodes. Each node projects and scan-converts its dataset partition, generating ray-segments equivalent fragments. As in [Ma95], ray segments are redistributed between nodes according to their image-space position. After a depth-sort, each fragment list is then composited. Furthermore, each node partitions its data subset into an identical copy of a kD-tree, in order to implement viewing volume culling and to reduce fragment list growth, as each leaf is processed in serial order according to its distance to the viewpoint.

Another hybrid method has been developed by Farias et al. [FMS00]. It uses a sweeping plane algorithm to compute an approximate object-space visibility ordering. As cells are encountered by the plane, they are projected onto the viewing plane and rasterized into fragments, which are inserted into pixel lists. Pixel list sorting provides an image-space correct ordering. The parallel implementation partitions the image-space between the N rendering nodes and maintains an octree storing the whole dataset. Each node determines the octree leaves intersecting the viewing volume portion relevant to each of its image-space tile. Then, it executes the sweeping algorithm on the data subset contained into the intersected leaves, producing a depth-correct image. As no fragments are redistributed between nodes, minimal communication is required. More recently, Childs et al. [CDM06] have presented a parallel algorithm suited to the rendering of very large datasets (100 million tetrahedra) on a PC cluster. They used a hybrid approach, similarly to [MC97]. By deferring large-element rasterization to a further stage, they were able to bound the workload performed by each node during each of the two phases, allowing quasi-linear scalability up to 400 nodes. Furthermore, by using a 3D-grid sampling approach during the object-space phase, they alleviated the need for sorting during the image-space phase, the compositing order being given by sample ordering along the grid depth axis. However this approach might lead to sampling artifacts and underestimation of the contributions of elements whose depth extent is smaller than the grid depth step size.

Several GPU image-space depth-sorting algorithms have been developed in the recent years. Callahan et al. [CIC*04] implemented a hybrid object-space/image-space unstructured volume rendering algorithm using a fixed-depth sorting network: the k-buffer. During a first phase, a partial object-space visibility ordering is determined on the CPU, by sorting tetrahedron faces according to their distance to the viewing plane. During the following GPU rasterization phase, fragments (depth and interpolated scalar value) are stored, for each pixel, into the k-buffer, which is implemented using k image-size 2D buffers and a fragment program. As a new fragment is inserted, the two closest fragments are determined, a ray segment contribution is computed and the closest fragment is then taken off the list. A correct image-space visibility ordering is computed as long as the difference in fragment position between partial and correct orderings does not exceed k. This depends on the dataset and the viewing position. Kipfer et al. [KSW04] implemented a particle system rendering engine on the GPU. In order to perform visibility ordering, which is necessary to correctly render semi-transparent particles, they developed a GPU-based bitonic

sorting algorithm. By taking advantage of the GPU's multiple fragment processing units, their algorithm was able to sort between 1 and 2 million particles per second, depending on the optimizations used.

3. CPU-GPU hybrid unstructured meshes parallel volume rendering – distributed architecture

In this section, we describe our distributed parallel unstructured volume rendering approach. As in [MC97], we use two main overlapping stages (Figure 1), interlinked by a N to P communication phase:

- the object-space stage, performed on the CPU, during which tetrahedron are projected, and scan-converted into ray segments,
- and the image-space stage, executed on the GPU, during which ray segments are depth-sorted and composited to produce final pixel values.

These two stages are pipelined in such a way that the projection stage performs processing of the i-th frame while the sorting/compositing stage computes the (i-1)-th frame. Moreover, each stage is executed in parallel with respectively N projection processes and P compositing processes.

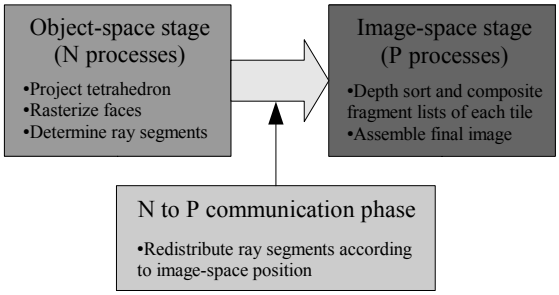

Figure 1: *Parallel rendering pipeline.*

Before actual execution of the rendering loop, the tetrahedral dataset is arbitrarily partitioned between the N projection processes. Each subset may have arbitrary geometry, topology, and/or connectivity, e.g. irregular boundaries, holes, or multiple disconnected components. Our only constraint is that no pair of cells should intersect, otherwise rendering artifacts might occur. During the object-space stage, ray segments are written in incoming order into a ray segment buffer, stored into main memory, which is a set of $W \times H$ lists, W and H being the image's dimensions. Each projection process maintains its own ray segment buffer into its memory space (Figure 2).

After tetrahedron projection ends, ray segments lists must be distributed to the compositing processes according to their image-space location. We use the MPI library in order to implement communication between the two stages. For each compositing process, a given projection process sends segment lists data relevant to the image-space part managed by the destination process. Each sorting/compositing process performs for each of its tile an image-space sorting of the assembled ray segments lists using the GPU, as in [CIC*04]. However, our sorting algorithm, based upon [KSW04], operates on the whole fragment list relevant to a given pixel, to compute a correct depth-ordering.

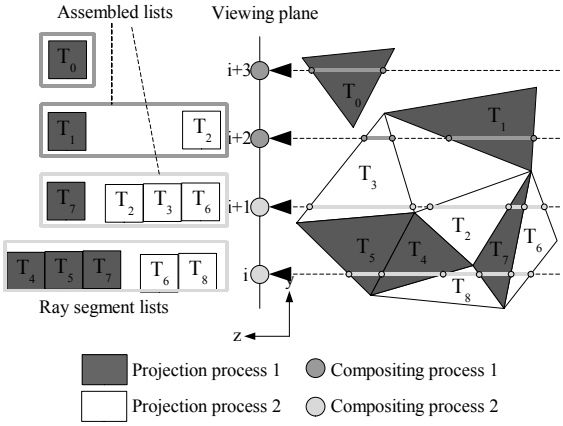

Figure 2: *Mesh and image-space partitioning.*

We use a simple tile-based image-space static subdivision scheme (Figure 3). Let n_W and n_H be the number of tiles respectively subdividing the image-space x and y axes. In order to attribute tiles to compositing processes, we scan the whole tiling from left to right in ascending scanline order. For each sequence of P tiles, we randomly shuffle the $\{0,1,...,P-1\}$ index sequence. The i-th tile in the sequence will be attributed to the process corresponding to the i-th shuffled process index.

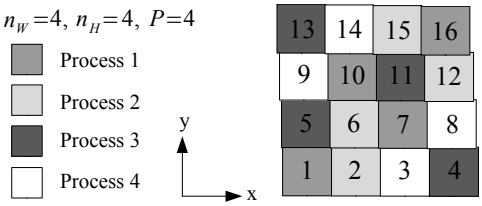

Figure 3: *Image-space tile-based partitioning.*

All tiles are evenly sized, with tile width $w_{tile}=W/n_W$ and tile height $h_{tile}=H/n_H$. The final image is assembled from the $n_W \times n_H$ tiles after the parallel compositing stage is over. Each compositing process sends its $n_W \times n_H / P$ tiles to the first compositing process, which assembles the final image according to the global tile layout. Note that this is the only sequential phase. In the following section, we describe in further detail the two main stages of our architecture.

4. CPU and GPU stages implementation details

4.1 CPU-based tetrahedron projection

We now present an overview of the tetrahedron CPU projection stage. The projection processes project and rasterize each of their tetrahedra in a sequential way, as described by the following pseudo-code:

Project the 4 vertices onto the viewing plane
Clip tetrahedron against viewing plane
Compute eye coordinates vertices-viewing plane distance
Determine tetrahedron face orientation relative to the viewing direction
Rasterize back faces
    Store each fragment into image-sized buffer B
Rasterize front faces
    Fetch corresponding back-face fragment from buffer
    Determine ray-segment from fragment pair
    and store into ray-segment buffer

Each tetrahedron vertex has two associated attributes: z the computed distance to the viewing plane and s the scalar field value (Figure 4). We use an orthographic projection for vertices. The projected tetrahedron is then clipped in order to trivially reject it if its bounding box falls outside of the viewing plane. For each face, orientation relative to the viewing direction is determined. Depending on the result, each projected face is classified either as a back-face or a front-face. For any pixel covered by the tetrahedron's projection, there is an associated front-face/back-face fragment pair, corresponding to the entry and exit points of a ray originating from the covered pixel.

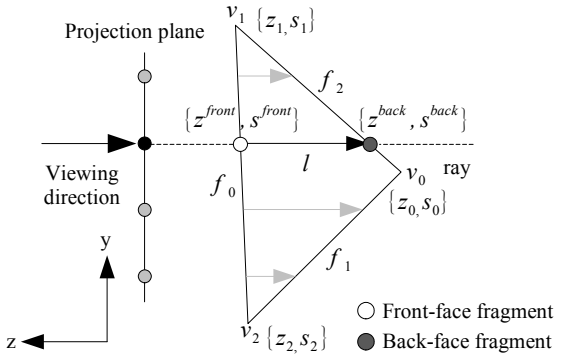

Figure 4: *Tetrahedron projection and ray segment determination.*

Each fragment pair allows to define a ray segment, which is characterized by a 4-tuple $\{z^{back}, l, s^{front}, s^{back}\}$, where z^{back} is the eye coordinates depth of the ray exit point, l the ray segment length, and (s^{front}, s^{back}) scalar values at the segment's extremities. These values are computed during face rasterization by linear interpolation between per-vertex values. Ray segments are stored into the ray-segment buffer according to their screen-space position: for every $p(i, j)$ pixel a list $L(i, j)$ stores segments which project onto $p(i, j)$. Ray segments are added to the relevant lists in incoming rasterization order. After tetrahedron projection is over, ray-segment redistribution occurs during the communication phase.

4.2 CPU SIMD face rasterization

Our SIMD-enhanced face rasterization algorithm processes four pixels at a time in order to reduce the number of rasterization steps. It uses x86 SSE instructions [Int05] implemented with C intrinsics. We define $\mathrm{V_i}$ as a 4-component vector storing four interpolated values at iteration i and $\{v\}_4$ as a vector where v is replicated in all 4 components. All vector operations performed during rasterization use SIMD instructions. The following pseudo-code describes our face triangle rasterization algorithm:

Sort vertices according to increasing y-coordinate
Determine edges orientation (o_0, o_1, o_2) *(left or right)*
Compute integer y-coordinate face extent (y_{min}, y_{max})
Rasterize all three edges
Rasterize scanlines

The sorted face vertices {min, mid, max} define three associated edges, rasterized according to the following order: {min, mid}, {mid, max} and {min, max} (Figure 5). Each edge is defined by vertices (v_{start}, v_{end}), being respectively the lower and higher y-coordinate vertices. Each vertex has two associated attributes $A = \{s, z\}$, which are its unprojected two attributes, and are linearly interpolated along the edge during rasterization:

For each edge (v_{start}, v_{end}) $k < 3$
    Determine edge array according to o_k
    Compute endpoints differences: $\Delta x = x_{end} - x_{start}$, $\Delta y = y_{end} - y_{start}$ *and* $\Delta A = A_{end} - A_{start}$
    Compute inverse slope $m_{inv} = \Delta x / \Delta y$
    Compute lower and higher scanline integer y-coordinates $y_a = \lceil y_{start} \rceil$ *and* $y_b = \lceil y_{end} \rceil$
    Compute y_p *and* $x_p = m_{inv} y_p$ *pre-step values*
    Compute pixels number $nb_{pix} = y_b - y_a$
    Compute iteration number: $nb_{it} = nb_{pix}/4$ *if* nb_{pix} *is a multiple of 4,* $nb_{it} = nb_{pix}/4 + 1$ *else*
    Compute interpolation factor $\mathrm{If_0}$ *and* $\mathrm{X_0}$ *vectors*
    For each 4-pixel interval $i < nb_{it}$
        Interpolate attributes $\mathrm{A_i} = \{A_{start}\}_4 + \mathrm{If_i}\{\Delta A\}_4$
        Store $\mathrm{A_i}$ *and* $\mathrm{X_i}$ *into edge arrays at* $y_a + 4i$
        Update interpolation factor $\mathrm{If_{i+1}} = \mathrm{If_i} + \{4/\Delta y\}_4$
        Update x-coordinate vector $\mathrm{X_{i+1}} = \mathrm{X_i} + \{4 m_{inv}\}_4$

Pre-step values allow to perform interpolation with subpixel accuracy. At each iteration i, the interpolated attributes $\mathrm{A_i}$ and x-coordinate $\mathrm{X_i}$ vectors corresponding to 4 edge-scanline intersections are computed. Attribute computation requires one addition and one multiply per attribute whereas x-coordinate and interpolation factor vector update both require only one addition.

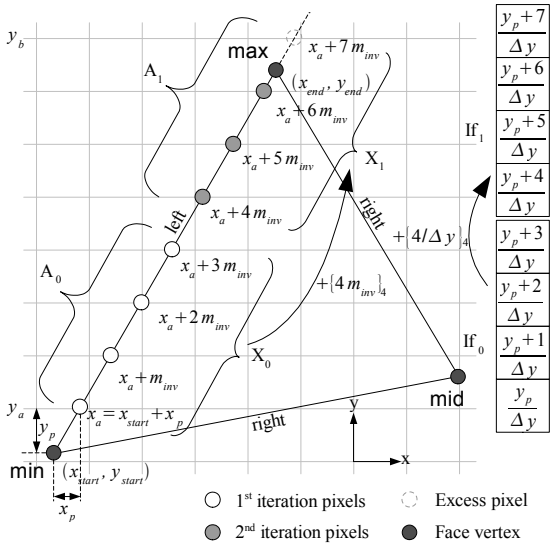

Figure 5: *SIMD triangle edge rasterization.*

At the end of each step, the three computed vectors are written into their corresponding 1D array. After the three edges have been rasterized, the left and right three 1D arrays contain x-coordinate, distance and scalar values for each span defined by two endpoints v_l and v_r (Figure 6). Scanlines are then rasterized:

For each scanline i *from* y_{min} *to* $y_{max}-1$
  Get x_l, x_r, *and attributes* A_l, A_r *from edge arrays and initialize per-line values*
  For each 4-pixel scanline interval $j<nb^i_{it}$
    Interpolate attributes $\mathrm{A_j}=\{A_l\}_4+\mathrm{If_j}\{\Delta A\}_4$
    Back face:
      Unpack $\{s^k, z^k\}$ *from* $\mathrm{S_j}$ *and* $\mathrm{Z_j}$
      Store nb^j_f *fragments into buffer B*
    Front face:
      Get nb^j_f *fragments from back fragments buffer*
      Pack $\mathrm{Z_j^{back}}$ *and* $\mathrm{S_j^{back}}$ *vectors*
      Compute ray length $\mathrm{L_j}=(\mathrm{Z_j^{back}}-\mathrm{Z_j^{front}})\{1/l_e\}_4$
      Pack ray segments $\{z^{back}_k, l_k, s^{front}_k, s^{back}_k\}$
      Store nb^j_f *segments into ray-segment buffer*
    Update interpolation factor $\mathrm{If_{j+1}}=\mathrm{If_j}+\{4/\Delta x\}_4$
    Decrement nb^i_{pix} *by 4*

As in edge rasterization, integer coordinates, endpoint attributes differences ΔA, interval pixel number nb^i_{pix}, interpolation factor If_0 and iteration number nb^i_{it} are computed prior to the rasterization loop (Figure 6). Each iteration requires one addition and one multiply per attribute interpolation and one addition to update the interpolation factor. In order not to overwrite or read excess fragments at step j, the effective fragment number is computed as $nb^j_f=4$ if $nb^i_{pix}>4$, else, as $nb^j_f=nb^i_{pix}$ which occurs only at the last step.

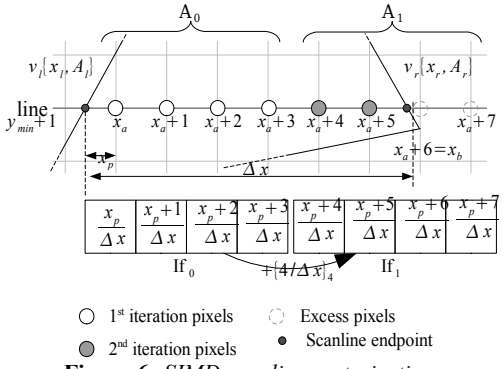

Figure 6: *SIMD scanline rasterization.*

During back face rasterization, the scalar value and depth $\{s^k, z^k\}$, $0\le k<4$, are unpacked and then written to the back fragments buffer B. During front face rasterization, nb^j_f ray segments are packed and written into the segment buffer. First, nb^j_f fragments are read from the back fragments buffer. Ray segment normalized length is then computed using $1/l_e$, which is the precomputed inverse length of the largest tetrahedron edge in the whole dataset. The four segments $\{z^{back}_k, l_k, s^{front}_k, s^{back}_k\}$ are packed from back fragments and the $\mathrm{Z_j}$, $\mathrm{S_j}$ and $\mathrm{L_j}$ vectors.

4.3 GPU image-space depth-sorting and compositing

We present here the GPU stage of our architecture, implemented with the OpenGL library. GPU compositing processes process their associated image-space tiles sequentially. Each tile is itself composited on a per-scanline basis:

For each associated image-space tile in the tile list
  For each tile line
    Send unsorted ray segment lists to the GPU
    Perform depth-sort in decreasing distance order using z^{back} *as a sorting key*
    Composite (back-to-front) sorted ray segment lists and output line pixel values

Before the main tile composition loop, each GPU process has received all the data corresponding to the unsorted segment lists. Remind that as shown in Figure 2, ray segments associated to the same pixel in image-space may have been generated by different projection processes. Thus, for a given tile k, the total unsorted segment list $L(i,j)$ corresponding to pixel (i,j), where $0\le i<w_{tile}$ and $0\le j<h_{tile}$, is obtained by assembling the N lists $\{L_0(i,j),\ldots,L_{N-1}(i,j)\}$ generated by the N projection processes. Each $L(i,j)$ list contains $S(i,j)$ elements.

We implemented the multi-pass GPU bitonic sorting algorithm developed by Kipfer et al. [KSW04]. As represented in Figure 7, the $w_{tile}\times h_{tile}$ element lists of a

given tile are sorted on a per-scanline basis. As any given scanline j is w_{tile} pixels wide, there are w_{tile} associated fragment lists $L(i, j)$, stored into a 2D sorting buffer in order to be sorted. The fragment list $L(i, j)$ therefore corresponds to the i-th sorting buffer line. The sorting buffer dimensions are (D, w_{tile}), where D is equal to the maximum hardware texture size (typically 4096 on current graphics hardware). As fragment lists are sorted independently, the maximum size S_{max} of a given list is equal to D. Thus, the maximum number of ray segments associated to any image-space pixel (that is, its allowable maximum depth complexity) is equal to D.

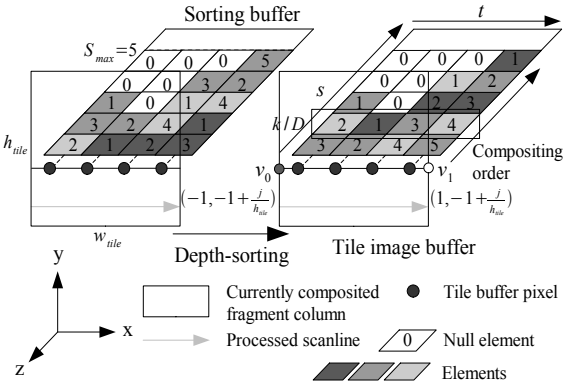

Figure 7: *Depth-sorting and compositing. Element color and number indicate CPU process origin and depth order.*

As in [KSW04], we use a ping-pong rendering technique with two different 2D buffers, respectively used to read elements output by the previous pass and output the current pass results. The following pseudo-code describes our implementation of GPU-based bitonic sorting.

Compute maximum list length $S_{max}=max\{S(i,j)\}$
Compute power-of-two length $S'_{max}=2^{\lfloor \log_2(S_{max}) \rfloor+1}$
Compute sorting phase number $nb_{phase}=\log_2(S'_{max})$

Fill input buffer with unsorted lists
Complete lists up to S'_{max} *with null elements*

For each sorting phase $m<nb_{phase}$
  For each sorting step n (from n = 0 to m)
    Perform bitonic sorting pass on the GPU
  Swap input and output buffers

During each pass, a number of 2D quads are rendered to span the whole output buffer. The span width is equal to S'_{max}, which is S_{max} rounded up to the nearest power-of-two, as the GPU bitonic sorting algorithm requires power-of-two length lists. Lists are completed with null elements $\{0,0,0,0\}$ up to S'_{max} before sorting. A bitonic sorting pass essentially compares and shuffles elements pairs, depending on *m*, *n* and element position within its list. We refer to [KSW04] for a full description. After sorting, null elements will be put at list end, as they have $z^{back}=0$.

At the end of sorting execution, the input buffer contains the w_{tile} sorted lists corresponding to scanline j. The corresponding final pixel values are then determined by our compositing algorithm, using the aforementioned buffer as an input, specified as a 2D texture). A $w_{tile}\times h_{tile}$ framebuffer object is used to store composited pixel values; for a given line j, any pixel i corresponds to the i-th sorted ray segment list stored into the input buffer. Compositing proceeds by accumulating all the input buffer's fragment columns contributions from back to front into the compositing buffer line (Figure 7).

For each input buffer element column $k<S_{max}$
  Rasterize line (v_0, v_1) *with associated texture coordinates* $(s_0, t_0)=(k/D, 0)$ *and* $(s_1, t_1)=(k/D, 1)$

Per-vertex coordinates (s,t) are linearly interpolated by the GPU in order to associate input buffer (k,i) element to line fragment (i,j). The s coordinate, which depends on k, addresses the element column whereas the t coordinate addresses the element within the column (see Figure 7). Each fragment's color and transparency contributions are computed by a fragment processing program. The composited element is looked up from the input buffer using the interpolated texture coordinate. As previously written, each sorting buffer element is defined as a 4-component vector $\{s^{front}, s^{back}, l, z^{back}\}$. Color and transparency are determined from the first three components by computing a numerical approximation of the emission/absorption optical model volume integral [Max95]. It is performed by executing a loop over the number of samples N_s, fixed for every fragment:

Set accumulated contribution to {0, 0, 0, 1}
Look up input buffer element using 1st texture coordinate
Compute step length $dx=l/N_s$

For each sample $p<N_s$
  Interpolate $s(p)$ *between* s^{back} *and* s^{front} *values*
  Look up color and optical density $(c(p),\rho(p))$ *from transfer function texture using* $s(p)$
  Compute sample color and transparency
  $C_p=c(p)dx$, $\Theta_p=e^{-\rho(p)dx}$
  Accumulate: $I=I+C_p\Theta_p$, $T=T\Theta_p$
Write result to output

For each sample p, the associated scalar value $s(p)$ is computed by linear interpolation. Then, the corresponding color and optical density values $(c(p),\rho(p))$ are read from a 1D RGBA floating-point transfer function texture with a dependent look-up using $s(p)$. The sample contributions are then computed and accumulated into the total contribution using the back-to-front compositing equations. At the end of the loop, the final ray segment contribution values (I,T) are output. After output,

fragment contributions are accumulated into the associated pixel using OpenGL floating-point alpha-blending. This is consistent with a tile framebuffer using a 16-bit per RGBA component pixel format. The blending mode used corresponds to back-to-front compositing.

As described above, lists are filled with null elements from $S(i,j)$ up to S_{max}. As each null element is defined as the $\{0,0,0,0\}$ vector, it ensures that the associated contribution has no effect on the accumulated final pixel values. As $l=0$, the final color contribution will be 0, as $C_p=c(p)\times 0=0$ and the final transparency contributions will be 1, as $\Theta_p=e^{-0}=1$.

5. Performance results

In this section, we present performance evaluation results of our rendering architecture on an 8-node PC cluster. Each node is a dual-processor Intel 3.4 GHz Xeon, with 8 Gbytes RAM, and a PCI-Express x16 Nvidia Quadro FX 4400 graphics board with 512 MB RAM. All the nodes are interconnected with a Gbit Ethernet switch, via a Gbit Ethernet network IC. Each node has the Red Hat Enterprise Linux Release 4 installed as an operating system and the 8174 Nvidia drivers version.

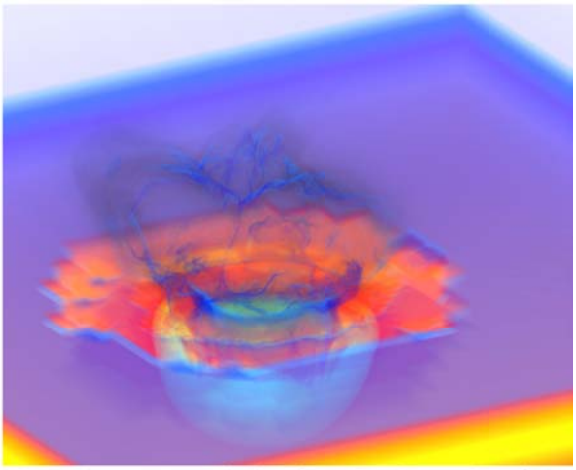

Figure 8: *15.7M tetrahedra dataset rendering (8 nodes).*

We evaluated aggregated rendering performance, that is, the total number of tetrahedron rendered per second, as a function of node number. We used two datasets with respectively 15.7 (Figure 8) and 51.1 million tetrahedra. These datasets represent the impact of a meteor into the ocean, computed with an adaptive meshing scheme. For each single measurement, a sequence of 50 images was rendered. We varied the number of nodes from 2 to 8, each node running one tetrahedron projection process and one depth-sorting and compositing process, the total number of processes therefore ranging from 4 to 16. A 1024x1024 image resolution was used, with a fixed 16x16 tiling. Due to memory restrictions, we couldn't perform tests on only one node, as the memory footprint of the two processes exceeded 8 GB. We therefore chose the 2 nodes results as a reference in order to compute parallel rendering efficiency.

Dataset	Node number	2	4	8
15.7 Mtets	kTetrahedra/s	878	1572	2533
	Efficiency	1	0.89	0.72
51.1 Mtets	Ktetrahedras/s	1021	1861	3406
	Efficiency	1	0.91	0.83

Table 1: *Parallel Rendering performance.*

At 8 nodes, we observe that parallel efficiency is 0.72 for the 15.7 Mtetrahedra dataset whereas it is equal to 0.83 for the larger one (Table 1). We also performed a detailed evaluation of the different steps of the two main stages in order to evaluate their impact on the overall rendering time for the 15.7 Mtetrahedra dataset. For both projection and sorting/compositing processes, we measured the average execution time over the total number of nodes. For each step of the overall execution, we also measured its average execution time over the total number of nodes, the sum of the steps average execution times being equal to the total stage execution time, for both kinds of processes.

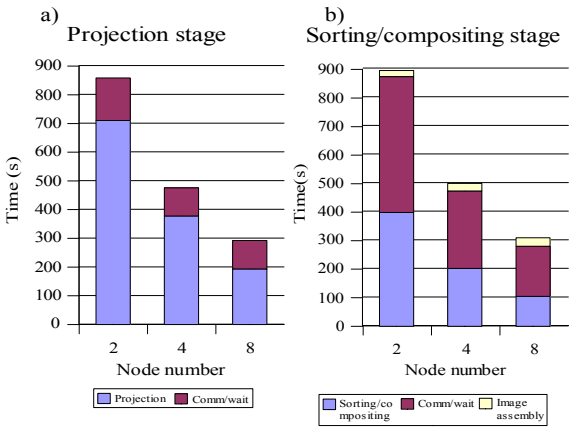

Figure 9: *CPU and GPU stages breakdown.*

Figure 9 a) represents the breakdown of the average projection time as a function of node number whereas Figure 9 b) shows the detailed breakdown of the average compositing process execution time. Concerning the projection stage, we observe that the average projection time makes up for the largest part of the overall execution time. Furthermore, it appears to decrease as a function of the inverse node number. However, we observe that as the number of nodes increases, the communication and wait phase duration makes up for a bigger part of the overall projection execution time. Similarly, a detailed examination of the sorting/compositing stage shows that the sorting and compositing execution time appears to decrease a a function of the inverse node number.

However, the average communication and wait time makes up for at least half the overall compositing execution time (about 56% for 8 nodes).

By comparing the projection and compositing stages breakdowns, we can conclude that our rendering architecture is limited by the projection stage, as for any node number the average projection time is greater than the sorting and compositing phases average execution time. On the average, compositing processes spend a large part of their communication/wait phase to wait for the projection processes to end their projection phase. The average final image assembly time is expected to remain constant as node number increases, as it is performed by only the first compositing process, receiving and assembling all the image tiles, while the other compositing processes send their image tiles to the first process. We actually observe that it slightly increases as node number grows: this may be caused by communication inefficiencies, as a larger number of processes send their tiles to the first compositing process, even if the total amount of transmitted data remains constant.

6. Conclusion and future work

As discussed in 4.3, the GPU-based depth-sorting algorithm does not allow per-pixel fragment lists being larger than the maximum texture size, and thus cannot correctly render datasets with a maximum depth complexity larger that the maximum texture size. We intend to modify and enhance it in order to support per-pixel arbitrary-length fragment lists. As shown by detailed performance breakdowns, the CPU projection phase is the current bottleneck of our parallel rendering architecture. We intend to take advantage of current multi-core CPUs in order to decrease the CPU projection phase time. Using multithreading with n_t threads per projection process, n_t being equal to the number of cores per CPU, we could ideally expect a n_t speedup.

In order to reach the target performance of 1 frame rendered/s with the 15.7M tetrahedra dataset, we would need at least 8x15.7/2.5=50 nodes, extrapolating the 8-node results from Table 1 and supposing that parallel efficiency does not further decrease. However, we think that the communication/wait phase duration could be reduced by using an interconnect system with a higher bandwidth than Gbit Ethernet, which would probably improve the overall rendering performance.

Finally, we would like to perform rasterization entirely on the GPU rather than on the CPU, as cell projection is at least one order of magnitude faster on the former. It would require being able to store the ray-segment buffer into graphics memory, which might be allowed by the latest GPU architectures.

7. Acknowledgments

We would like to thank Philippe Ballereau for the meteor datasets. We would also like to thank the reviewers for their helpful remarks.

References

[CIC*04] CALLAHAN S. P., IKITS M. , COMBA J. L. D., SILVA C. T.: Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (May 2005), 285-295

[CDM06] CHILDS H., DUCHAINEAU M., MA K.-L.: A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In *Proc. of the Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 153-162.

[FMS00] FARIAS R., MITCHELL J. S. B., SILVA C.T.: ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering, In *Proc. of the 2000 IEEE Symposium on Volume Visualization* (2000), pp. 91-99.

[Int05] INTEL: IA-32 Intel Architecture Software Developer's Manual, Basic Architecture, April 2005.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: UberFlow: A GPU-Based Particle Engine. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware* (2004), pp. 115-122.

[Ma95] MA K.-L.: Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures. In *Proc. Of the IEEE symposium on Parallel Rendering* (1995), pp. 23-30.

[MC97] MA K.-L., CROCKETT T.W.: A Scalable Parallel Cell-Projection Volume Rendering Algorithm for Three-Dimensional Unstructured Data. In *Proc. of the IEEE Symposium on Parallel Rendering* (1997), pp. 95-104.

[Max95] MAX N.: Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (June 1995), 99-108.

[MCE*94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, (1994), 23-31.

[SMW*04] STRENGERT M., MAGALLON M., WEISZKOPF D., GUTHE S., ERTL T.: Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters, In *Proc. of the Eurographics Symposium on Parallel Graphics and Visualization* (2004), pp. 41-48.