Eurographics Symposium on Parallel Graphics and Visualization (2013)
F. Marton and K. Moreland (Editors)

Practical parallel rendering of detailed neuron simulations

J.B. Hernando [1], J. Biddiscombe [2], B. Bohara [3], S. Eilemann [3], F. Schürmann [3]

[1]CeSViMa, Universidad Politécnica de Madrid, Spain, [2]CSCS, Swiss National Supercomputing Centre, Lugano, Switzerland,
[3]Blue Brain Project, École Polytechnique Fédérale de Lausanne, Switzerland

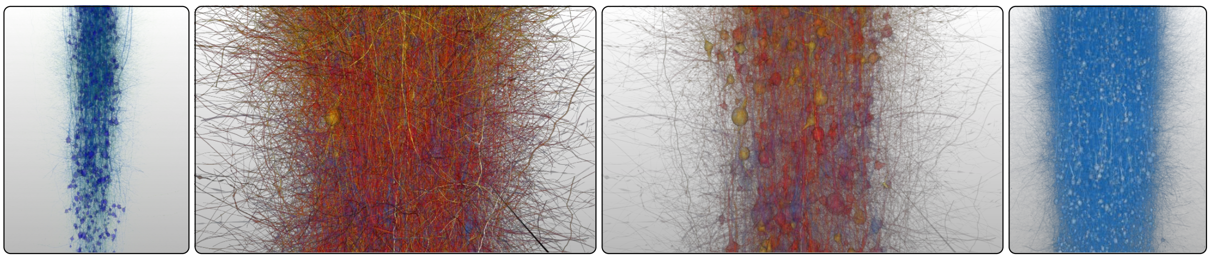

Figure 1: *Four different renderings of a subset of the cortical circuit, from left to right: Full circuit view of 1000 neurons with thickness dependent transparency, close-up of 2000 neurons with simulation data and activity dependent transparency, the same view with additional thickness opacity modulation, 5000 neurons with transparency.*

Abstract

Parallel rendering of large polygonal models with transparency is challenging due to the need for alpha-correct blending and compositing, which is costly for very large models with high depth complexity and spatial overlap. In this paper we compare the performance of raster-based rendering methods on mesh models of neurons using two applications, one of which is specifically tailored to the neuroscience application domain, the other a general purpose visualization tool with domain specific additions. The first implements both sort-first and sort-last and uses a scene graph style traversal to cull objects, and dual depth peeling for order independent transparency, whilst the other uses a simpler brute force data-parallel approach with sort last composition. The advantages and trade offs of these approaches are discussed.

We present the optimized algorithms needed to achieve interactive frame rates for a non-trivial, real-world parallel rendering scenario. We show that a generic data visualization application can provide competitive performance when optimizing its rendering pipeline, with some loss of capability over an optimized domain-specific application.

Categories and Subject Descriptors (according to ACM CCS): I.3.m [Computer Graphics]: Miscellaneous—Parallel Rendering Computer Applications [J.3]: Life and Medical Sciences—Biology and genetics;

1. Introduction

Simulation-based research has become an effective tool for scientific discovery in many areas. Supercomputers not only follow Moore's law per processors, but continually increase the number of processors per system, causing a data explosion above the industry trend. Furthermore, many of the simulation domains are inherently three-dimensional in nature, which calls for interactive 3D applications for debugging, analysis, discovery and communication of scientific results.

The mammalian brain is a densely packed structure. The geometrical model of 5,000 cortical neurons, which occupy a few mm^3 of brain cortex, has in excess of 720×10^6 triangles. Due to their geometrical properties they are challenging to render, and transparency is needed to reduce visual clutter, making visualizations of neuronal tissue simulations a challenging problem for interactive parallel rendering.

We compare two different approaches to tackle these rendering challenges: an optimized scenegraph-based application versus ParaView with enhancements for the given problem. We present a quantitative analysis of how much interactivity can be gained for this problem by using a custom tailored code compared to a readily available parallel ren-

DOI: 10.2312/EGPGV/EGPGV13/049-056

dering software. The custom tailored application, RTNeuron, is based on OpenSceneGraph and Equalizer. It implements both sort-first and sort-last parallelization [MCEF94], and any combination of them. Sort-first profits from the view frustum culling technique presented in [HPS12], whereas sort-last uses a novel spatial partitioning for mesh data with fast, RGBA-only compositing detailed in Section 3.1.2.

We present rendering performance results for several circuit sizes, rendering modes, simulation data mapping and different parallel rendering strategies, run on a medium-sized GPU cluster. Our results show that parallel rendering for real-world scenarios requires careful tuning of the rendering pipeline. While ParaView provides competitive performance with careful optimizations, a specialized application performs similarly using a data model not optimized for rendering, but designed to enable domain-specific functionality.

2. Related work

Parallel rendering concepts, algorithms and systems have been well studied previously. Our work builds on the ParaView visualization application [Hen07] and RTNeuron, an optimized application using the OpenSceneGraph [RO*13] and the Equalizer parallel rendering framework [EMP09].

ParaView is a widely used scalable parallel visualization tool based on VTK [SML03a] that makes use of IceT [MKPH11] for image compositing. Whilst it already supports sort-last parallel rendering including transparency, the default implementation of data distribution and compositing is unsuitable for the very large and complex models used in this study. A number of improvements that have been made to improve interactivity are discussed in Section 3.2.

In the area of parallel rendering, significant work has been published for parallel compositing, load balancing, data distribution, architecture and general scalability. From this work, a few generic frameworks emerged, including Chromium [HHN*02], ClusterGL [NHM11], OpenSG [VBRR02], VR Juggler and derivatives [BJH*01], OpenGL Multipipe SDK [BRE05], Equalizer [EMP09] and CGLX [DK11]. We use Equalizer as the basis for the optimized application due to its flexibility in configuration, feature set and maturity.

To optimize the costly recomposition for sort-last rendering, a number of parallel compositing algorithms have been proposed [MPHK94, LRN96, SML*03b, EP07, PGR*09, MEP10]. Further compositing optimizations include image compression [AP98, YYC01, SKN04] and screen-space bounding regions [MPHK94, YYC01]. Exploiting the NUMA topology of hybrid GPU clusters has been shown to improve performance of GPU-based applications [SMV11, EBA*12]. We use direct-send sort-last compositing since it trivially allows ordering all images during composition, and since message contention [YWM08] is not a bottleneck in the cluster sizes used for our problem set. Furthermore, we enable real-time RLE compression, application-provided region of interest and NUMA-aware thread affinity, as described in detail in [EBA*12].

3. Parallel rendering of neuronal tissue simulations

The models used in this study are of similar build to those reported in [HWR*12], resembling a functional cortical column of the somatosensory cortex of a young rat. We use digitized and postprocessed 3D neuron models and place 10,000 of them in a 3D volume of about 0.5×1.5 mm (diameter$\times$height) according to biological constraints. To prepare the simulations, neuron arbor overlaps are identified, and a portion of those locations are turned into functional synapses. The neurons themselves are functionalized using the multi-compartment Hodgkin-Huxley formalism following methods described in [DBG*07, HHS*11]. The typical biophysical observable reported per compartment and time step is the membrane voltage, which for post-mortem visualization is dumped into a large list of scalars sorted by compartment per neuron, neuron number and time step, respectively. Depending on the scientific question, also transmembrane current or spike times may get reported in addition to or instead of the membrane voltage. The simulations of the full model were run on a 4-rack BlueGene/P supercomputer using the software NEURON [CH06, HES08].

Individual neurons are represented as a tree of conical frustra (segments) encoding branching structure, diameter and direction changes, called *morphological skeletons*. For visualization, triangular surface meshes are created according to [LHS*12]. For out data set, the neurons average 4,200 segments or 140,000 triangles per neuron, totaling upwards of 1.4×10^9 triangles for a 10,000 neuron column. Two scenarios are considered; one where a base set of several hundred different meshes is reused at different locations and another where all the meshes are unique.

Individual neurons are geometrically complex tree-shaped objects with an uncommon aspect ratio: they have a very large bounding volume compared to the space filled with geometry, which causes sub-pixel geometry and aliasing problems. Neurons are entwined and tightly packed in cortical circuits, leading to abundant occlusion and visually very cluttered scenes. These features call for elaborate visual metaphors and visual analytics techniques for truly insightful visualizations. There is not a single optimal way of representing these datasets and we believe that real-time rendering of the raw simulation results in full detail is needed for initial exploration and as the foundation for finding more appropriate visual designs with the help of neuroscientists.

Our current visualizations focus on the presentation of simulation results mapped onto the detailed neuron surface, but still providing capabilities to interactively change visual attributes like color mapping, visibility and representation style (e.g. *soma* (neuron center) only or whole neuron) for individual neurons. To deal with occlusion and clutter we use selective transparency by modulating the opacity of neuronal branches or by mapping simulation values to the geometry, mimicking wet lab techniques such as voltage sensitive dyes.

The rendering of cortical circuits combines high geometrical complexity with expensive transparency rendering. The latter requires sorting geometry at the pixel level, which is a tremendous computational load due to the high depth complexity of our data sets. Parallel rendering become a necessity to achieve interactive framerates and to enable rendering of large circuits using sort-last decomposition.

3.1. RTNeuron: an optimized application

Our customized application, RTNeuron, is a GPU accelerated rendering application based on OpenSceneGraph for data management and Equalizer for parallel rendering. It focuses not only on fast rendering times, but also in fast loading times with no offline preprocessing. It provides level of detail (LOD) rendering, high quality anti-aliasing based on jittered frusta and accumulation during still views, interactive modification of the visual representation of neurons in a per-neuron basis (full neuron vs. soma only, branch pruning depending on the branch level, ...).

The application provides two different levels of detail for neurons. The high LOD uses fully detailed meshes, while the lower LOD uses raycast spheres and Phong-shaded pseudo-cylinders. The pseudo-cylinders are rendered using a geometry shader to convert lines into screen aligned quads and per fragment shading to provide the cylindrical appearance. Transparent rendering uses dual depth-peeling [BM08].

The implementation puts a special emphasis on keeping separate data structures for each neuron, so different shader parameters and LOD selection can be applied to each individual neuron. This has implications on the scenegraph layout and view frustum culling, as described in Section 3.1.1.

3.1.1. Sort-first parallelization

For geometry intensive applications, scalable sort-first can only be achieved by means of effective view frustum culling (VFC) and accurate load balancing. Two reasons make VFC hard in our case: First, we have densely packed and entangled objects which are very concave (in the mathematical sense). Second, we need to keep the rendering primitives for each object independent, so different rendering parameters can be easily applied.

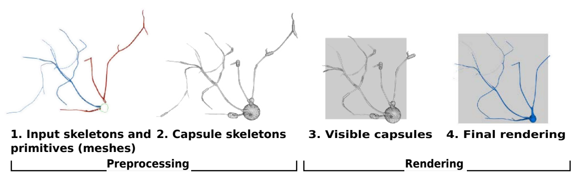

Figure 2: *Overview of view frustum culling in RTNeuron*

Figure 2 shows how VFC is performed in RTNeuron. The key element of the algorithm is the *capsule skeleton*, a list of capsules directly obtained from the morphological skeleton of the neuron. Each capsule bounds a piece of branch and maps to a range in the primitive index list (which is sorted to traverse each branch as orderly as possible). Visibility tests consist of frustum-capsule intersection tests, followed by collapsing the primitive ranges of those visible to find the polygons to render. The set of visible capsules can be computed using a geometrical data partition such as an octree or a k-d tree, but given the regularity of the data structure for each neuron (a plain array of capsules), it is also possible to solve each frustum-capsule intersection test in parallel using the GPU. In this work both the GPU approach (implemented using CUDA) and the octree based algorithm have been evaluated. Results are presented only for the CUDA algorithm as it performs consistently better in almost all cases. The benefits of this approach are several. Once the capsules have been tested for visibility, the rendering step is $O(nm)$, n being the number of capsule per neuron and m the number of visible neurons. This minimizes the number of rendering API calls, thanks to the orderly arrangement of capsules and primitive lists and the collapsing of capsule ranges, while keeping each object independent as required. The disadvantage of using the CUDA culling is that the scenegraph becomes flat. As the number of objects in the scene grows this can become a problem for culling, geometry dispatching and depth peeling as discussed in section Section 4.1. The reader is referred to [HPS12] for further details about the view frustum culling algorithms.

3.1.2. Sort-last parallelization

Our sort-last decomposition is based on a k-d tree partition using a point cloud approximation of the real circuit to minimize memory usage during setup. The point cloud can be generated using the global position of the segment midpoints from the morphological skeletons. Neurons are processed in batches to keep memory usage bounded. This point cloud produces a balanced partition for pseudo-cylinder models, but is imbalanced for meshes. To improve it, each morphological point is weighted by the number of mesh vertices that have it as it closest one.

The k-d tree construction is run by each processor independently taking as inputs the processor count n and the aforementioned point cloud $\mathcal{C}$. No processor actually builds the whole tree, but only the path from the root to its leaf. At each subdivision step, the split plane position λ is computed based on the ratio of processors assigned to each subspace, which translates into a quantile value of the point distribution along the split axis. The quantile value can be computed exactly in $O(n)$ time using the SELECT algorithm [BFP*73], but since $\mathcal{C}$ is already an approximation of the rendering load, we instead find λ by binning.

At the end of the algorithm each processor stores the list of split planes and the final bounding box of the k-d tree leaf assigned to it. During rendering, the bounding box is used to setup hardware clipping planes, and the plane list is used to find the view dependent compositing order.

The algorithm carries an array of neuron identifiers along $\mathcal{C}$ to map points to neurons and decide which models need to be finally loaded in each processor. During loading, the models are postprocessed in two ways depending on whether unique or shared morphologies are used: For *shared morphologies*, neurons are clipped reusing the mechanisms of VFC. The capsule skeletons are postprocessed to annotate the capsules that are outside the tree leaf. The annotation is used during the primitive range collapse to mask out these capsules. For *unique morphologies*, the capsule skeletons and the geometric primitives for all LODs are postprocessed and clipped to the leaf bounding box.

In both cases, the final number of objects to be processed per frame will be in the order of magnitude of the full circuit; at least for a circuit size with the extent of a cortical column, because a large portion of the neurons is still present in each k-d tree leaf. Nonetheless, with shared morphologies, the maximum amount of geometry to render by each process is bounded and the depth complexity of the scene reduced (which is important for transparency). With unique morphologies, the actual size of the dataset is greatly reduced compared to sort-first for the same processor count. For sufficiently small leaf volumes it will be even less than for shared morphologies because the geometry cannot be clipped in the latter case.

The final compositing step is handled by the Equalizer backend. In Equalizer, the compositing phase is specified declaratively in a configuration file. We used direct send, which allows easily for an arbitrary compositing order at runtime, needed for fast and correct compositing of transparent renderings. Thanks to the k-d tree partition and given the properties of direct-send compositing, there is no need to exchange depth buffer information between the processes. An additional optimization is the use of *regions of interest* to limit the screen area to be read back and transmitted to that where the k-d tree leaf projects. This is particularly useful for full circuit views with high processor counts.

3.2. ParaView: integration in a standard tool

Whilst the ParaView visualization application handles large polygonal meshes with ease, performance drops considerably when transparent geometry is rendered and it proved to be unsuitable for the models used here. The problems may be summarized as follows:

- Independent per-vertex opacity is not supported. 1D color tables may have arbitrary RGBA values, but 2D lookup tables with independent RGB and A components from different scalar arrays are not supported for meshes.
- Transparent blending is limited to depth peeling with a fixed number of peels and not available on all hardware.
- Partitioning of data in parallel is handled automatically when transparency is requested, but does not share spatial decomposition information with polygonal data readers/filters which may generate pre-partitioned data.
- The data partitioning step converts data to unstructured grids and then back to polygonal meshes at render time.

The first issue prevents visualization of the kind desired, the latter issues result in poor performance and excessive memory consumption. To improve the capabilities and rendering speed, a number of enhancements have been made which are described as follows.

3.2.1. Partitioning

When geometry is rendered with transparency, ParaView instantiates a distributed data filter which partitions data between processes such that each process owns a region that is one leaf of a k-d tree. The compositing order, passed to IceT, is generated by traversing the tree to generate the ordering that ensures correct back-to-front compositing of individual regions in the same way as Section 3.1.2.

This inbuilt data distribution suffers from limitations; the redistribution is performed as a final stage before rendering, which means that data produced by a reader or other filter which changes over time, will trigger a re-execution of the redistribution even when the geometry is in fact static. The problem arises because the pipeline does not distinguish between static and dynamic geometries – an obvious solution is to enhance the pipeline with this information to allow better discrimination. However, to support time dependent scalars with fixed decompositions, it is preferable to cache and reuse this information at the reader end of the pipeline – an arbitrary number of filters may be inserted between source and sink making it hard to guarantee that caching is honoured. ParaView provides a mechanism to pass parallel *extent* information down the pipeline when working with structured/uniform grid datasets, we have extended this feature to work with polygonal datasets by providing a custom *extent translator* object that stores the spatial bounds of each partition and the overall k-d tree by attaching them to each piece of the neuron data passed through the processing chain. Only operations which alter the parallel piece distribution can invalidate this information and it is therefore easy to avoid accidental corruption. At the rendering end of the pipeline it is only necessary to insert a simple check to see if a k-d tree has been supplied with the data, and if so, skip the (automatic) redistribution and pass the tree directly to the IceT compositing engine.

Neuron data loading uses the same library as RTNeuron, reading neuron meshes passing them as triangle lists to VTK where they can be partitioned. For efficiency reasons, we use the Zoltan [DBH*02] library for this purpose. It has been integrated into a *vtkMeshPartitionFilter* used to partition the neurons using an optimized Recursive Coordinate Bisection (RCB) method based on the algorithm described in [BB87]. This produces evenly distributed meshes across processes and the generated k-d tree can be used to identify the process regions, as discussed previously. Our filter integration in the Zoltan library preserves the data type of the VTK mesh and thus saves unnecessary duplications of cells.

3.2.2. Painters

The rendering framework of ParaView uses specialized painters, each of which performs a specific task and delegates other tasks to other painters. The default painter chain comprises the following major components (some omitted for brevity): *ScalarsToColors*$\rightarrow$ *ClipPlanes*$\rightarrow$ *Lighting*$\rightarrow$ *Primitive*. To add per-vertex opacity and per-vertex colors, we replaced the *ScalarsToColors* painter with a *TwoScalarsToColors* painter which accepts two arrays, mapping one through a color table and adding the other as the transparency value (direct or via lookup table).

Correct blending is implemented using depth peeling or back-to-front sorting. The high depth complexity of the models and fixed number of peels in the ParaView implementation, as well the option of using CPU only machines ruled in favour of a depth sorting approach. We therefore implemented a *DepthSort* painter to generate a camera distance based ordering (without copying or modifying cells) and a *SortedPrimitive* painter to render the cells using the ordered array passed down to it, resulting in the following pipeline: *TwoScalarsToColors*$\rightarrow$ *DepthSort*$\rightarrow$ *ClipPlanes*$\rightarrow$ *Lighting*$\rightarrow$ *SortedPrimitive*.

Cells lying on process boundaries may overlap the bounding boxes of the regions in which they reside. This may lead to a wrong draw order causing artefacts, however the triangles are extremely small compared to the full scene and are not visible unless the camera is zoomed into a closeup view of the overlap zone. Due to the fact that the meshes are well formed non-intersecting surfaces, mutually overlapping triangles are not present and the painter's algorithm produces correct images within the process bounds.

3.2.3. GPU Acceleration

The painter chain presented in the previous section enables the generation of depth correct images, but does trigger a modified state for each frame due to reordering of cells. This prevents the use of a *DisplayList* painter to accelerate the OpenGL rendering, causing the performance to be orders of magnitude worse than possible. Recent work on frameworks such as Piston [LSA12] makes it easier to combine CPU/GPU processing within ParaView, and when combined with Cuda-GL interoperability, offers the possibility of reusing GPU objects directly in the rendering phase. By performing the depth sorting of cells on the GPU and storing the vertex and color arrays in place on the GPU for rendering, we can completely eliminate the per-frame CPU to GPU data transfer, dramatically increasing performance over the CPU sorting mode.

When GPU rendering is enabled, we remove the *DepthSort* painter from the chain, and replace the *SortedPrimitive* painter with a new *PistonPolygons* painter. The piston-based painter implements polygons sorting using the Thrust library [BH11]. It pushes the VTK dataset (when modified) to the GPU prior to rendering, creating vertex and color arrays using Thrust device arrays. The render phase includes the cell sorting algorithm, which generates an element array of sorted IDs. This array, along with the vertices and color-table-mapped RGBA values are then passed directly to OpenGL for rendering.

4. Experiments

To evaluate the performance and scalability of the different rendering approaches we have run several experiments on a fat node cluster with the following technical specification: dual six-core 3.47GHz processors (Intel Xeon X5690), 24GB of RAM and three NVidia GeForce GTX580 GPUs (3GB RAM); 10Gbit/s ethernet and 40Gbit/s QDR InfiniBand. The GPUs are attached each using a dedicated 16x PCIe 2.0 link, the InfiniBand on a dedicated 8x PCIe 2.0 link and the 10 Gbit/s Ethernet on a 4x PCIe 2.0 link. For both applications we have used up to 6 cluster nodes (which equals to 18 GPUs) and rendered 30 frames at a 1920×1200 screen resolution. CPU only benchmarks were run on a Cray XK7 with 272 nodes of 2.1Ghz 16-core AMD Opteron chips. In the software stack, the operating system is RHEL 6.3 with an x86_64 2.6.32 Linux kernel. The OpenGL NVidia driver version is 310.32, we have verified that this version does improve performance considerably compared to versions 270 and 295. OpenSceneGraph 3.0.1 and the latest Equalizer and Collage source codes (git tag EGPGV13) have been used. The ParaView base version is 3.98 using mpich-3.0.2 on the GPU cluster and Cray mpich2-5.6.1 on the XK7. The experiments consist on the rendering of:

- Circuits with sizes: 1,000 ($1K$), 2,000 ($2K$) and occasionally 5,000 ($5K$). The circuit sizes are 14.5×10^6 triangles / 4.3×10^6 segments for $1K$, 30×10^6 triangles / 8.7×10^6 segments for $2K$ and 720×10^6 triangles / 21×10^6 segments for $5K$.
- with shared neuronal morphologies or simulated unique morphologies,
- rendered with opaque and transparent materials,
- using a full-circuit camera and a closeup camera,
- with and without simulation playback (RTNeuron only)

Figure 1 shows combinations of these variables. Not all combinations have been tested for each application, the exceptions are given below.

4.1. Strong scaling

4.1.1. RTNeuron

The experiments have been performed using dynamic and static sort-first as well as static sort-last. Dynamic sort-first relies on Equalizer to adjust the viewports reactively based on previous frame rendering times. Sort-first uses horizontal stripes, as it provides better results than vertical stripes or

2D tiles. Two variations of the camera positions have been used: static and slowly moving. The latter triggers the VFC code every frame, while the former computes the visibility only once for the static decompositions. In these experiments only shared morphologies have been used.

We have observed serious performance issues when using three GPUs with multi-threaded sort-first rendering (for all driver versions tested), and therefore resorted to multi-process execution (one per GPU), which performs as expected but at increased memory usage.

The results are presented in Figure 3. In general, sort-first performs better than sort-last at the resolution chosen. Transparency equalizes the results and close-ups are better handled by sort-first as expected. Static sort-first partitions perform on par to dynamic sort-first in many cases, but this just a consequence of the viewpoints chosen.

Reasonable scalability is observed for full circuit viewports up to 3 nodes (9 GPUs). From then on, additional GPUs provide only a slight improvement. Overlap between tiles in sort-first and compositing overhead for sort-last partially explain this, but even more important is fragmentation of rendering dispatching. Due to object independence and neuron's shapes, with finer partitions more rendering calls of fewer elements are issued for each neuron. With close ups, the overhead of VFC and fragmentation becomes more apparent (notice the difference between `SSf*` and `DSf` for $2K$). Scalability at interactive framerates for the opaque rendering is challenging for all these reasons. These results are consistent with previous findings [EBA*12].

In transparency there is a bottleneck associated with the number of objects in the scene, which makes the rendering for $2K$ more than twice as slow compared to $1K$. The plots also show how the cost of VFC is amortized by the passes required by dual depth peeling. Some atypical results that require further inspection are the performance of 2 nodes for closeups with transparency and the high variance observed in sort-last for 5 and 6 nodes for $1K$ opaque neurons, in special for closeup views.

Experiments with simulation playback and the LOD based on pseudo-cylinder were also performed. For simulation, the plots have very similar profiles. For pseudo-cylinders, the reduced memory usage allows to use $5K$ neuron circuits as well. The scalability in these cases is worse because there is the same VFC overhead and number of rendering calls whilst the polygon count per call is much smaller. Also, sort-last outperforms sort-first in this cases. These plots can be found in the additional material.

4.1.2. ParaView

For the ParaView implementation we performed fewer tests, but they compare well to the equivalent cases for RTNeuron. The two key factors for good performance are that the depth sort operation is $O(n \log_2 n)$ complexity (parallel merge sort) and that data is loaded and partitioned as a single dataset per

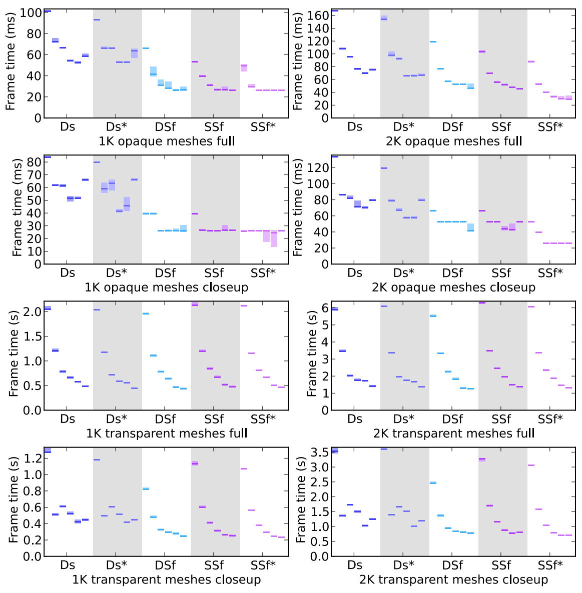

Figure 3: *Strong Scaling: median and inter-quartile range for RTNeuron frame times without simulation. Each colored group shows the rendering times for 1 to 6 nodes (3-18 GPUs) for a different parallelization strategy (DS: direct-send, DS*: direct-send still camera, DSF: load-balanced sort-first, SSF: static sort-first, SSF*: static sort-first still camera).*

process, which results in large flat arrays for the mesh data elements (vertices, scalars, attributes, ...). When passed to the GPU, the triangles are rendered in a single draw operation which maximizes throughput, in contrast to RTNeuron which keeps objects separate to allow per-object operations. Performing the sort and clip operations entirely on the GPU results in a higher degree of parallelization than would occur on CPUs since the number of CPUs per node is small in this case.

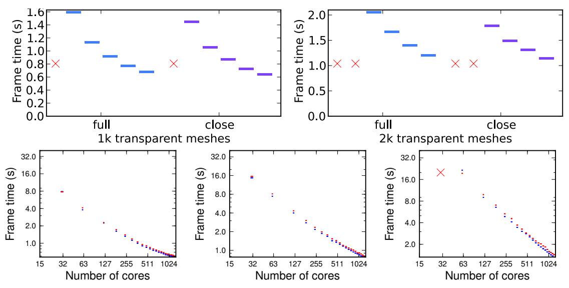

Figure 4: *ParaView timing for GPU, 1-6 nodes (top) and CPU (bottom full, close) rendering. Data points marked × are unavailable due to memory constraints.*

Figure 4 shows times for $1K$, $2K$ and $5K$ circuits rendered on the GPU cluster (1-6 nodes) and on a CPU cluster. Due to memory requirements of the load balancing stage some timings are not available (denoted by ×). Both circuits scale well to 18 GPUs achieving a render time of $\approx 0.7s$ for $1K$ neurons and $\approx 1.2s$ for $2K$ neurons. The close-up views do not show a significant speedup relative to full views as clipping is a relatively cheap operation and takes place after sorting. One possible optimization may be to clip first and thus save time during the sort phase. Software rendering times on CPUs are slower, however, there is almost no size limit to the model that can be handled (given sufficient resources). On the CPU, close up views show a slight slowdown relative to full views as more pixels are being filled. When combined with LOD rendering for interaction, the results show that it is possible to work with very large models in full detail.

4.2. RTNeuron weak scaling

Weak scaling tests with unique morphologies have been run using 1 to 18 GPUs with VFC performed every frame. The circuit size has been chosen as a function of the GPU count n, using $500 \times n$ for opaque rendering and $400 \times n$ for transparency (which is more memory limited). Figure 5 shows the frame times measured for different configurations. Additionally, Figure 6 shows the minimum and maximum of total neurons and triangles loaded by each process. The plots show that the geometrical workload is fairly stable and balanced, however the number of neurons increases as expected.

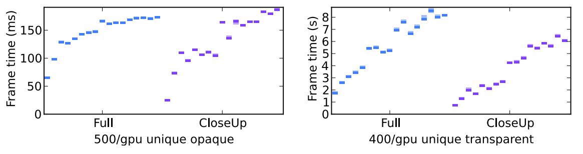

Figure 5: *Weak scaling: Median rendering time for opaque (500 neurons per GPU) and transparent (400 neurons per GPU) meshes using unique geometries and a closeup view.*

At similar geometrical workloads per GPU, having more objects in the scene affects rendering speed for different reasons in each case: In opaque renderings the problems are the GPU-based VFC, which has to deal with an increasing amount of smaller objects, and draw call fragmentation as mentioned in Section 4.1.1. With transparency, VFC can be amortized over the multiple peel passes; the main issue instead is the increasing number of peel passes (the same polygon count in less volume implies higher depth complexity).

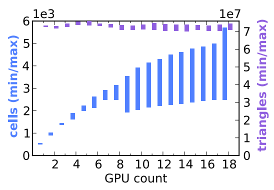

Figure 6: *Minimum and maximum neuron and triangles counts assigned to each GPU in the weak scaling tests from Figure 5.*

5. Conclusions and future work

Despite the different approaches to the problem, the experimental results show that the performance of both RTNeuron and the ParaView solution is similar for transparent renderings. In RTNeuron, domain specific features are easier to implement, it is more memory conservative (using sort-last) and slightly faster rendering times can be achieved in certain cases as well. The trade-offs are that the ParaView-based tool provides better scalability as the circuit size grows thanks to its data layout, it can be used *in-situ* with CPU rendering and total development costs are higher for RTNeuron.

In our workflows, parallel rendering proves to be a valuable tool to enable interactive framerates for transparent rendering. For opaque rendering, performance scaling is less important due to the relatively fast rendering speed, but it is a critical enabler to render larger circuits using weak scaling.

We plan to optimize the scalability by addressing the bottlenecks found in our extensive experiments. For RTNeuron a long-standing feature is application-driven predictive load balancing in Equalizer and VFC can be optimized by pipelining the algorithm with the drawing. Dynamic load balancing for sort-last will be explored to address the depth complexity problem identified in Section 4.2. For ParaView we plan to handle ghost cells on process boundaries to remove artefacts and to add support for some of the alternative visualization options available in RTNeuron.

6. Acknowledgments

This work was supported in part by the Blue Brain Project, the Swiss National Science Foundation under Grant 200021-116329/1 and by the Spanish Ministry of Science and Innovation under grant (TIN2010-21289-C02-01/02) and the Cajal Blue Brain Project.

References

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization* (1998). 2

[BB87] BERGER M. J., BOKHARI S. H.: A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput. 36*, 5 (May 1987), 570–580. 4

[BFP*73] BLUM M., FLOYD R. W., PRATT V., RIVEST R. L., TARJAN R. E.: Time bounds for selection. *J. Comput. Syst. Sci. 7*, 4 (Aug. 1973), 448–461. 3

[BH11] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems, Jade Edition, Edited by Wen-mei W. Hwu* (2011). 5

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A virtual platform for virtual reality application development. In *Proceedings of IEEE Virtual Reality* (2001), pp. 89–96. 2

[BM08] BAVOIL L., MEYERS K.: *Order Independent Transparency with Dual Depth Peeling*. Tech. rep., NVIDIA Corporation, 2008. 3

[BRE05] BHANIRAMKA P., ROBERT P. C. D., EILEMANN S.: OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization* (2005), pp. 119–126. 2

[CH06] CARNEVALE N. T., HINES M. L.: *The NEURON Book*. Cambridge University Press, 2006. 2

[DBG*07] DRUCKMANN S., BANITT Y., GIDON A. A., SCHÜRMANN F., MARKRAM H., SEGEV I.: A novel multiple objective optimization framework for constraining conductance-based neuron models by experimental data. *Frontiers in Neuroscience 1*, 1 (2007). 2

[DBH*02] DEVINE K., BOMAN E., HEAPHY R., HENDRICKSON B., VAUGHAN C.: Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering 4*, 2 (2002), 90–97. 4

[DK11] DOERR K.-U., KUESTER F.: CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics 17*, 2 (March 2011), 320–332. 2

[EBA*12] EILEMANN S., BILGILI A., ABDELLAH M., HERNANDO J., MAKHINYA M., PAJAROLA R., SCHÜRMANN F.: Parallel rendering on hybrid multi-gpu clusters. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 109–117. 2, 6

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics* (May/June 2009). 2

[EP07] EILEMANN S., PAJAROLA R.: Direct send compositing for parallel sort-last rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2007). 2

[Hen07] HENDERSON A.: *ParaView Guide, A Parallel Visualization Application*. Kitware Inc., 2004-2007. 2

[HES08] HINES M. L., EICHNER H., SCHÜRMANN F.: Fully implicit parallel simulation of single neurons. *Journal of Computational Neuroscience 25*, 3 (aug 2008), 439–448. 2

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics 21*, 3 (2002), 693–702. 2

[HHS*11] HAY E., HILL S., SCHÜRMANN F., MARKRAM H., SEGEV I.: Models of Neocortical Layer 5b Pyramidal Cells Capturing a Wide Range of Dendritic and Perisomatic Active Properties. *PLoS Comput Biol 7*, 7 (07 2011), e1002107. 2

[HPS12] HERNANDO J. B., PASTOR L., SCHÜRMANN F.: Towards real-time visualization of detailed neural tissue models: view frustum culling for parallel rendering. In *BioVis 2012: 2nd IEEE Symposium on biological data visualization* (2012). 2, 3

[HWR*12] HILL S. L., WANG Y., RIACHI I., SCHŘMANN F., MARKRAM H.: Statistical connectivity provides a sufficient foundation for specific functional connectivity in neocortical neural microcircuits. *Proceding of the National Academy of Science 109*, 42 (09 2012), E2885–94. 2

[LHS*12] LASSERRE S., HERNANDO J., SCHÜRMANN F., DE MIGUEL ANASAGASTI P., ABOU-JAOUDÉ G., MARKRAM H.: A neuron membrane mesh representation for visualization of electrophysiological simulations. *IEEE Transactions on Visualization and Computer Graphics 18*, 2 (2012), 214–227. 2

[LRN96] LEE T.-Y., RAGHAVENDRA C., NICHOLAS J. B.: Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (July-September 1996), 202–217. 2

[LSA12] LO L.-T., SEWELL C., AHRENS J.: Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV* (2012), pp. 11–20. 5

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (1994), 23–32. 2

[MEP10] MAKHINYA M., EILEMANN S., PAJAROLA R.: Fast compositing for cluster-parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 111–120. 2

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, ACM, pp. 25:1–25:10. 2

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 59–68. 2

[NHM11] NEAL B., HUNKIN P., MCGREGOR A.: Distributed OpenGL rendering in network bandwidth constrained environments. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2011), pp. 21–29. 2

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings ACM/IEEE Conference on High Performance Networking and Computing* (2009), pp. 1–10. 2

[RO*13] ROBERT OSFIELD D. B., ET AL.: OpenSceneGraph. http://www.openscenegraph.org/, 2001-2013. 2

[SKN04] SANO K., KOBAYASHI Y., NAKAMURA T.: Differential coding scheme for efficient parallel image composition on a pc cluster system. *Parallel Computing 30*, 2 (2004), 285–299. 2

[SML03a] SCHROEDER W. J., MARTIN K., LORENSEN W.: *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, Third Edition*. Kitware, Inc. (formerly Prentice-Hall), 01 2003. URL: `http://www.vtk.org`. 2

[SML*03b] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 33–40. 2

[SMV11] SPAFFORD K., MEREDITH J. S., VETTER J. S.: Quantifying NUMA and contention effects in multi-GPU systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 11:1–11:7. 2

[VBRR02] VOSSG., BEHR J., REINERS D., ROTH M.: A multi-thread safe foundation for scene graphs and its extension to clusters. In *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2002), EGPGV '02, Eurographics Association, pp. 33–37. 2

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings IEEE/ACM Supercomputing* (2008). 2

[YYC01] YANG D.-L., YU J.-C., CHUNG Y.-C.: Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing 18*, 2 (February 2001), 201–22–. 2