Sort-Twice Algorithms for Polygon Rendering with PC Clusters

Thomas Funkhouser, Kai Li, and Rudrajit Samanta
Princeton University

1 Introduction

The objective of our research is to investigate how to construct a high-performance and inexpensive parallel rendering system leveraging the aggregate performance of multiple commodity graphics accelerators in PCs connected by a system area network.

The main challenge is to develop efficient partitioning and load balancing algorithms that scale well within the processing, storage, and communication characteristics of a PC cluster. As compared to traditional, tightly-integrated parallel computers, the relevant limitations of a PC cluster are that the processors (PCs) do not have fast access to a shared virtual address space, and the bandwidths and latencies of inter-processor communication are significantly inferior. Moreover, commodity graphics accelerators usually do not allow efficient access through standard APIs to intermediate rendering data (e.g., fragments), and thus the design space of practical parallel rendering strategies is severely limited. The challenge is to develop algorithms that partition the workload evenly among PCs, minimize extra work due to parallelization, scale as more PCs are added to the system, run at interactive frame rates (30 fps), and work efficiently within the constraints of commodity components.

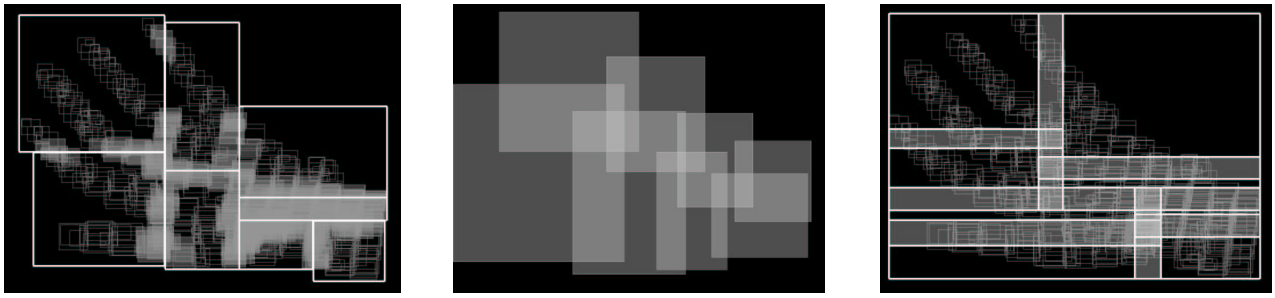
Over the last four years, we have developed several algorithms and systems for parallel rendering on PC clusters. Our initial system was built to drive a multi-projector display wall [8] – there we used a sort-first strategy with dynamic coarse-grained load balancing algorithms. More recently, we have investigated how to use a PC cluster to drive a single display without using special purpose hardware [5, 6, 7]. This paper reviews some of the main ideas behind this recent work: sort-twice partitioning [6] and k-way replication [5]. The following two sections describe these two approaches and analyze how well they work. The final two sections summarize experimental results achieved with a working prototype system and discuss future research directions.

2 Sort-Twice Partitioning

Parallel rendering algorithms can be classified by the stage at which data is sorted [3]. Recent work in cluster rendering has generally focused on sort-first and sort-last approaches. For instance, WireGL [1] uses a sort-first strategy to distribute rendering primitives simultaneously from multiple clients to multiple servers. Although this approach supports immediate-mode APIs (OpenGL), its efficiency (utilization of the graphics hardware) is poor because the bandwidth of cluster networks is at least an order of magnitude less than that of graphics cards. Also, the scalability of sort-first systems is limited by the increasing overheads of redundant rendering when primitives overlap multiple tiles. Alternatively, sort-last systems have been built for PC clusters [10], but they have high pixel composition bandwidth overheads that limit the frame rate and/or resolution of the system.

Our approach is to use a hybrid parallel rendering algorithm that combines features of both sort-first and sort-last strategies [6]. We call this approach "sort-twice." Like a pure sort-last system, we partition the 3D model into disjoint *groups* of polygons, render each group on a different PC, and use depth compositing to merge the resulting partial images into a final image for display. As in any sort-last system, each primitive is rendered exactly once, and there are no overheads due to redundant rendering. However, like a sort-first system, we partition the 3D model dynamically for each frame using a view-dependent algorithm that minimizes screen-space overlaps of primitives rendered on different PCs. After each server has rendered its primitives, we employ a peer-to-peer networking phase in which each server is responsible for compositing pixels for separate *tiles* of the screen. Since both 2D screen and 3D scene partitions are created together dynamically in a view-dependent context for every frame, the hybrid load balancing algorithm can create 2D tiles and 3D groups such that the region of the screen covered by any group of 3D polygons is closely correlated with the 2D tile of pixels assigned to the same PC. In this case, less processing and network bandwidth is required to transfer and composite pixels during each frame.

The advantage of this hybrid sort-twice approach over pure sort-first and pure sort-last is motivated by the example in Figure 1, which shows visualizations of the overheads incurred by different parallel rendering strategies while rendering a 3D model of a hand (more light gray roughly corresponds to greater overheads). Figure 1(a)



(a) Sort-first (b) Sort-last (c) Hybrid

Figure 1: Visualizations of overheads during parallel rendering of hand with different algorithms. In (a), highlighted object bounding boxes span multiple tiles and must be rendered redundantly. In (b) and (c), brighter pixel intensities represent more image composition overheads.

shows a pure sort-first partition of pixels into tiles (white lines), with primitive groups overlapping multiple screen tiles highlighted in gray. Figure 1(b) shows a pure sort-last partition of primitives, with pixels requiring composition highlighted in gray. Finally, Figure 1(c) shows a hybrid sort-first and sort-last partition. Note that primitives are rendered only once and pixel composition is required only for thin swaths of pixels at the boundaries of tiles.

During simulations with varying system parameters, we find that the sort-twice algorithm outperforms sort-first and sort-last algorithms in almost all tests, including ones with larger numbers of processors and higher screen resolutions. Figure 2 shows sample breakdowns of server processing times for each algorithm. On the leftmost set of bars ("Sort-First"), note the the dark bands ("Overlap Render") representing overheads due to redundant rendering of objects overlapping multiple tiles. On the rightmost set of bars ("Sort-Last"), note the light colored bands ("Pixel Read" and "Pixel Write") representing overheads due to pixel composition. In both cases, the overheads become a larger percentage of the total server time with increasing numbers of processors, which indicates limited scalability. The sort-twice algorithm ("Hybrid") largely avoids both types of overheads, thereby providing much better efficiency and scalability.

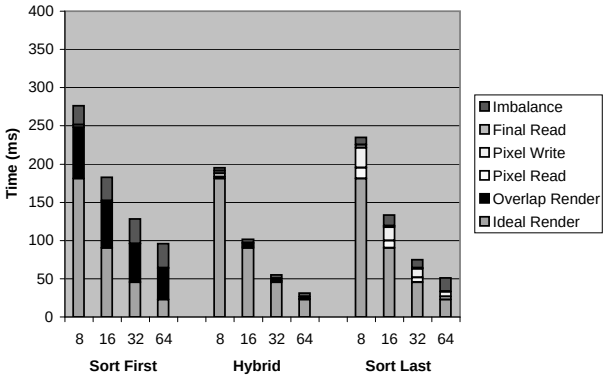


Figure 2: Server time breakdowns for sort-first, sort-last, and the hybrid algorithms for screen resolution 1280x960. The height of each bar represents the time required for processing in the server.

3 K-way Replication

Although communication overheads can be reduced by partitioning the workload in a view-dependent manner, a direct implementation of the sort-twice approach requires either replicating the entire 3D scene on every PC [6, 7] or dynamically re-distributing primitives in real-time as the user's viewpoint changes [4]. Unfortunately, neither approach is practical for a PC cluster, since the memory of each PC is usually too small to store all the data for a very large model, and the network is too slow to transmit 3D primitives between PCs in real-time.

Our approach is based on *k-way replication* of the scene data [5]. During an off-line phase, we organize the input 3D model into a multiresolution hierarchy of objects and replicate each object on k out of n server PCs ($k << n$). Then, during an on-line phase, we perform a view-dependent partition of the objects (as described in the previous section), selecting exactly 1 out of k servers to render each object. The key idea is to avoid replicating the entire 3D model on every PC and to avoid real-time transmission of 3D primitives, while achieving reduced communication overheads due to dynamic view-dependent partitioning.

The motivations for this k-way replication strategy are evident in Figure 3. The image on the left (Figure 3(a)) shows the partition for a pure sort-last system with 1-way replication (no copies)– it must composite nearly full-screen images if the primitives assigned to each processor are distributed uniformly throughout the model (processor assignments are indicated by color). Alternatively,

(a) k = 1 (pure sort-last) (b) k = 4 (partial replication) (c) k = 16 (full replication)

Figure 3: K-way replication ($k = 4$) enables view dependent partitioning without full replication. Color of each bounding box indicates which server renders its enclosed triangles.

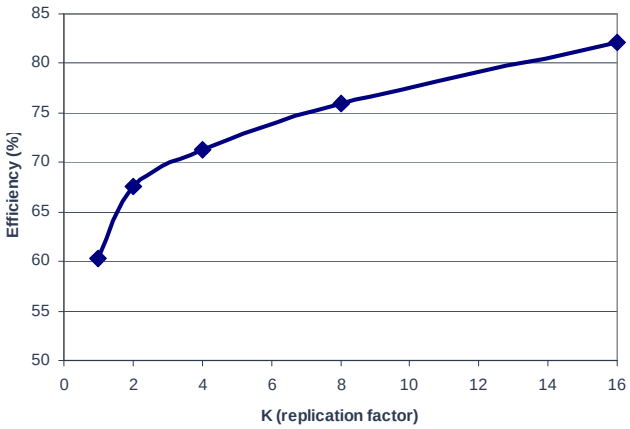


Figure 4: Plot of parallel rendering efficiency with varying replication factors (k). Note that the Y-axis begins at 50%.

a system with n-way replication (full replication) can reduce the image composition overheads by assigning primitives to processors dynamically for each view in order to minimize the size of screen regions rendered by different processors [6]. For instance, in Figure 3(c)), image composition is required only for the thin swaths at the seams between primitives of different colors. Unfortunately, this purely view-dependent partitioning approach requires the entire scene to be replicated on every processor. The k-way replication approach avoids full replication of the scene data but still can employ a view-dependent load balancing algorithm, since every primitive is available on more than one processor. With k-way replication, we are able to construct partitions similar to those of n-way replication but with storage costs closer to 1-way replication (see Figure 3(b)).

Figure 4 shows results of simulations aimed at evaluating the trade-offs of k-way replication in a sort-twice parallel rendering system. As the replication factor (k) increases, the system's efficiency increases, but so too do the storage requirements. However, note that the efficiency improvement is non-linear, with most of the benefits of replication occurring for small values of k. As a result, we find that moderate replication factors (e.g., $k \leq 4$) provide almost all the benefit of full replication without all the cost.

4 Experimental Results

We have developed a prototype system that implements the methods described in this paper, and we use it to conduct experiments to evaluate performance. We use the system's efficiency (useful polygon rendering time divided by total frame time) at interactive frame rates (15 frames/second) as our primary metric for success.

Our experimentation platform is a PC cluster with a client PC, 24 server PCs, and a display PC. Each PC is a Dell Precision Workstation 420 with a 733Mhz Pentium III CPU, an Intel 840 chipset with 133Mhz front-side bus, 256MB of dual-channel RDRAM memory, and a nVidia GeForce-II chip based graphics card. Each PC runs Microsoft Windows 2000. The communication network is Myrinet. Each PC uses a previous-generation, 32-bit 33Mhz PCI network interface card that has 2MB of SDRAM and a 33Mhz LANai-4 network processor. The 26 PCs are networked together with a 32-port switch which is implemented with eight 8-port crossbar switches. We have used the GM driver for Windows 2000 provided by Myricom. The total cost of the system is around $50K.

We report results of experiments with the three test models shown in Figure 5. They were selected based on their complexities and details. Each one contains too much data to fit into the memory of a single PC, and they have surface details that motivate a user to zoom in and examine the models closely.

In every experiment, we logged performance statistics while the system rendered images for a camera moving along a simulated user's viewing path. Each path started with the camera framing the 3D model. It rotated around the model (1/2), zoomed up close to the surface, panned for a while (1/2), and then zoomed back out. Except for our final speedup results (in the last subsection), the multiresolution scene graph traversal was set to render around 100,000 polygons per frame on each server.

Figure 6 shows the results of an experiment testing the scalability of our system as we add more servers. We see that the system is able to achieve between 30M and 48M triangles/sec for the three test models in our maximum test configuration ($k = 6$ and $n = 24$). The performance includes the overhead of software image composition and

(a) Michelangelo's "David" [2] (8,254,150 triangles, ∼800MB)

(b) Face of Michelangelo's "St. Matthew" [2] (6,755,412 triangles, ∼700MB)

(c) Visible Man Skeleton (without feet) [9] (2,432,525 triangles, ∼250MB)

Figure 5: Test models used in our experiments.

the overhead to send result pixels to the display. This performance represents about 52.1%, 65.3%, and 73.9% efficiencies for St. Matthew, David, and Visible Man, respectively, while executing at 12.9, 16.25, 20.0 frames per second.

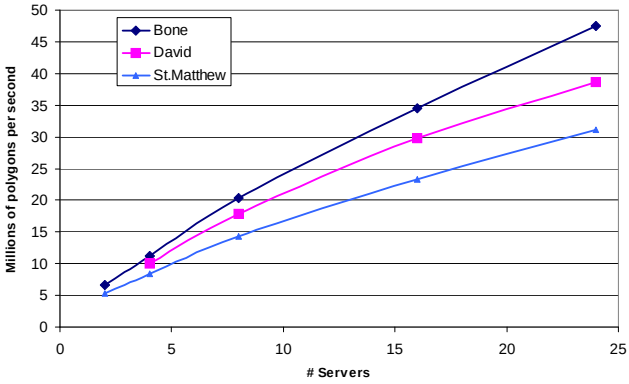


Figure 6: Plot of millions of polygons per second rendered by our real system during tests with three large models.

5 Conclusion

This paper describes sort-twice and k-way replication strategies for parallel rendering with a PC cluster. The key ideas behind sort-twice is to perform view-dependent partitions of both the 3D model and 2D screen space in order to reduce the overheads of pixel composition in a sort-last system. The key idea behind k-way replication is to provide a dynamic load balancing algorithm some ability to perform view-dependent partitioning while avoiding full replication of the scene data on every PC.

While these ideas may be promising, many problems remain open for future study and discussion. First, the methods proposed work only for static scenes. Second, the network requirements are currently suitable only for single screen resolutions. Third, the k-way replication strategy has not yet been implement for out-of-core rendering. Extending the proposed methods to handle these more difficult situations is a topic for future research.

References

[1] Greg Humphreys, Mathew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: A scalable graphics system for clusters. In *Computer Graphics (SIGGRAPH 2001)*, August 2001.

[2] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 131–144, 2000.

[3] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[4] Carl Mueller. Hierarchical graphics databases in sort-first. In *Proceedings of the IEEE Symposium on Parallel rendering*, pages 49–57, 1997.

[5] Rudrajit Samanta, Thomas Funkhouser, and Kai Li. Parallel rendering with k-way replication. *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, October 2001.

[6] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 99–108. ACM Press, August 2000.

[7] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Sort-first parallel rendering with a cluster of pcs. In *SIGGRAPH 2000 Technical sketches*, August 2000.

[8] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *SIGGRAPH '99. Proceedings 1999 Eurographics/SIGGRAPH workshop on Graphics hardware, Aug. 8–9, 1999, Los Angeles, CA*, pages 107–116. ACM Press, 1999.

[9] Greg Turk. Large geometric models archive. *www.cc.gatech.edu*, 2000.

[10] Brian Wylie, Constantine Pavlakos, Vasily Lewis, and Ken Moreland. Scalable rendering on pc clusters. *IEEE Computer Graphics and Applications*, 21(4):62–69, July 2001.