

# Parallel Rendering with K-Way Replication

Rudrajit Samanta, Thomas Funkhouser, and Kai Li

Princeton University\*  
Princeton, NJ 08540

## Abstract

With the recent advances in commodity graphics hardware performance, PC clusters have become an attractive alternative to traditional high-end graphics workstations. The main challenge is to develop parallel rendering algorithms that work well within the memory constraints and communication limitations of a networked cluster. Previous systems have required the entire 3D scene to be replicated in memory on every PC. While this approach can take advantage of view-dependent load balancing algorithms and thus largely avoid the problems of inter-process communication, it limits the scalability of the system to the memory capacity of a single PC. We present a k-way replication approach in which each 3D primitive of a large scene is replicated on  $k$  out of  $n$  PCs ( $k \ll n$ ). The key idea is to support 3D models larger than the memory capacity of any single PC, while retaining the reduced communication overheads of dynamic view-dependent partitioning. In this paper, we investigate algorithms for distributing copies of primitives among PCs and for dynamic load balancing under the constraints of partial replication. Our main result is that the parallel rendering efficiencies achieved with small replication factors are similar to the ones measured with full replication. By storing one-fourth of Michelangelo's David model (800MB) on each of 24 PCs (each with 256MB of memory), our system is able to render 40 million polygons/second (65% efficiency).

**Keywords:** Parallel rendering, interactive visualization, cluster computing, computer graphics systems.

## 1 Introduction

In spite of recent advances in computer graphics hardware, the demands for polygon rendering rates continually exceed the capabilities of any single computer. For example, isosurfaces extracted from The Visible Woman contain approximately 10 million polygons [14], and 3D models of Michelangelo's sculptures captured with laser-range scanners contain tens of millions, or even billions, of triangles [13]. In these cases and many others, the models are at least an order of magnitude too large to be rendered at interactive rates, and yet subpixel-resolution details of rendered images are important to the user, preventing use of detail elision techniques to accelerate frame rates [6]. For decades to come, or at least until

Jurassic Park can be rendered at interactive frame rates, the demand for high-performance graphics will exceed the capabilities of any single computer by a large factor.

Our objective is to build an *inexpensive* polygon rendering system capable of displaying complex 3D polygonal models at interactive frame rates. Most previous work in high performance rendering has focused on building large, high-end computers with multiple tightly-coupled processors, for example SGI's InfiniteReality and UNC's PixelFlow machines. The main drawback of this approach is that the hardware in these systems is custom-designed and therefore very expensive, sometimes costing millions of dollars.

The focus of our research is to investigate parallel rendering with inexpensive commodity components. In this study, we aim to construct an *efficient* and *scalable* system leveraging the aggregate rendering performance of multiple PCs connected by a system area network. The motivations for this approach are numerous. First, we can leverage the favorable price-to-performance ratio of commodity graphics hardware, which far exceeds that of custom-designed, high-end rendering systems. Second, we can readily take advantage of future technology improvements by replacing commodity components as faster versions become available. Third, we can scale the aggregate system bandwidth simply by adding more PCs to the network. Finally, we can employ general-purpose computers that can be used for other applications when not running visualization programs.

The main challenge is to develop efficient partitioning and load balancing algorithms that work efficiently within the processing, storage, and communication characteristics of a PC cluster. As compared to large-scale, tightly-integrated parallel computers, the two most important limitations of a PC cluster are that each processor has a limited amount of memory (e.g., 256MB), and cluster communication performance is typically an order of magnitude less than a tightly-coupled system (e.g., 100MB/sec vs. 1GB/sec). The challenge is to overcome these limitations by developing coarse-grained algorithms that distribute the storage of the model across multiple PCs, partition the workload evenly among PCs, minimize extra work due to parallelization, scale as more PCs are added to the system, and work efficiently within the constraints of commodity components.

Recent work in parallel rendering with PC clusters has focused on strategies that assign rendering work for different parts of the screen to different PCs [18, 24, 25]. It has been shown that communication overheads can be reduced by partitioning the workload in a view-dependent manner so that pixels rendered by one PC can be sent to the display directly with little or no depth compositing. However, current methods require either replicating the entire 3D scene on every PC [24, 25] or dynamically re-distributing primitives in real-time as the user's viewpoint changes [19]. 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. Since current trends indicate that the size of 3D models and the performance of 3D graphics accelerators are growing faster than PC memory capacity and network bandwidth, these problems will only become worse in the future.

---

\*[rudro,funk,li]@cs.princeton.edu

Our approach is based on partial replication of data. 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 ( $k \ll n$ ). Then, during an on-line phase, we perform a view-dependent partition of the objects, selecting exactly 1 out of  $k$  servers to render each object. We use a peer-to-peer, sort-last communication strategy to composite the rendered images to form the final image. 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 goal of our study is to investigate how well this approach works and to understand the tradeoffs of various algorithms at each stage of the process.

In this paper, we present a working prototype system that performs parallel rendering on a PC cluster without any special hardware. Our main contributions are: (1) a novel  $k$ -way replication strategy that trades off memory for communication overheads in a parallel rendering system, (2) new methods for distributing replicated primitives and assigning work to avoid starvation, maintain balance, and reduce overheads, and (3) integration of hierarchical scene traversal with load balancing to maintain interactive frame rates during parallel rendering of large scenes. Using these ideas, we have built a system that is able to render 30-48M triangles per second at interactive frame rates while storing only one-fourth of the scene in memory on each PC.

## 2 Background and Related Work

Parallel rendering systems are often classified by the stage in which primitives are partitioned: sort-first, sort-middle, or sort-last [7, 17].

Most traditional systems have been based on the sort-middle approach, in which graphics primitives are partitioned among geometry processors, screen tiles are partitioned among rasterization processors, and a fast communication network is used to send primitives from geometry processors to rasterization processors based on their tile overlaps. This approach is not well-suited for a PC cluster due to its high communication bandwidth requirements.

In sort-first systems, screen-space is partitioned into non-overlapping 2D tiles, each of which is rendered independently by a tightly-coupled pair of geometry and rasterization processors, and the subimages for all 2D tiles are composited (without depth comparisons) to form the final image. The main advantage of this approach is its relatively low communication requirements. Unlike sort-middle, sort-first can utilize retained-mode scene graphs to avoid most data transfer for graphics primitives between processors [18, 19]. For instance, Samanta et al. [25, 26] described a sort-first system in which a static scene database is replicated on every PC of a cluster. Buck, Humphreys et al. [4, 12] described a PC-based system in which one or more clients send OpenGL commands over a system area network to servers rendering different parts of the screen. In both cases, the efficiency is limited by the extra work that must be done to render graphics primitives redundantly if they overlap multiple tiles. In general, since overlap factors grow linearly with increasing numbers of processors, the scalability of sort-first systems is limited [18]. Moreover, since the rendering assignment of any graphics primitive to a processor is made dynamically based on its screen-space projection, assignments can vary rapidly. Thus, the scene database must be fully replicated for fast access by every processor, or a dynamic scene distribution algorithm must be used to move scene data from processor to processor based on its predicted screen projection. Either method limits the scalability of the system for a PC cluster.

In sort-last systems, each processor renders a separate image containing a portion of the graphics primitives, and then the resulting images are composited (with depth comparisons) into a single image for display. The main advantage of sort-last is its scalability.

Since each graphics primitive is rendered by exactly one processor, the overlap factor is always 1.0. A disadvantage of sort-last is that it does not preserve primitive ordering semantics, which makes antialiasing and transparency difficult to support. Also, sort-last systems usually require a network with high bandwidth for image composition. Although it is possible to build special-purpose networking hardware for compositing [16, 27, 30], this approach requires custom-designed components, which are expensive. Also, latency is induced if compositing is done in multiple stages [16].

There has been considerable work over the last decade on reducing the overheads of image composition in sort-last systems. For instance, Ma et al. [15] reduced pixel redistribution costs by keeping track of the screen-space bounding box of the graphics primitives rendered by each processor. Cox and Hanrahan [9] went further by keeping track of which pixels were updated by each processor (the “active” pixels) and only composited them instead of compositing the entire screen. Ahrens and Painter [1] compressed pixel color and depth values using run-length encoding before transmitting them for composition. Other researchers have investigated composition schemes for specific network topologies [15, 20, 29] or viewing conditions [21]. However, some of these methods are effective only if the viewpoint does not “zoom-in” to view a portion of the model, and none of them has achieved low enough communication bandwidths for interactive rendering on a PC cluster. Most encouraging speedup results have been reported for systems operating either at low resolution (such as 256x256 or 512x512) or at non-interactive frame rates (a few frames per second).

Recently, Samanta et al. [24] described a hybrid system in which sort-last pixel compositing overheads were significantly reduced by executing a dynamic sort-first primitive partition for each viewpoint. Since both pixel and primitive partitions were created together dynamically in a view-dependent context, their algorithm could create tiles and groups such that the region of the screen covered by any group of 3D polygons was closely correlated with the 2D tile of pixels assigned to the same PC. Accordingly, relatively little network bandwidth was required to re-distribute pixels from PCs that rendered polygons to others that composited the overlapped pixels. While this approach achieves attractive efficiency (55-70%), it still required full replication of the scene database on every PC, thereby limiting its scalability.

To summarize, none of these previous methods allows visualization of very large models while leveraging the cumulative rendering power of commodity PCs. Current sort-middle and sort-last systems require too much communication bandwidth. Current sort-first and hybrid systems require too much memory in every PC. Our goal is to develop a system that overcomes these limitations of a PC cluster and achieves low-cost, high-efficiency parallel rendering.

## 3 Basic Approach

Our architecture is based on the hybrid sort-first and sort-last approach of Samanta et al. [24]. The key new idea is to replicate every primitive in a scene  $k$  times among  $n$  rendering PCs (where  $k \ll n$ ). We call this strategy “ $k$ -way replication.”

The motivations for  $k$ -way replication are evident in Figure 1. In the image on the left (Figure 1(a)), a sort-last system with 1-way replication (no copies) 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 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 [24]. For instance, in Figure 1(c), image composition is required only for the thin swaths at the seams between primitives of different colors.

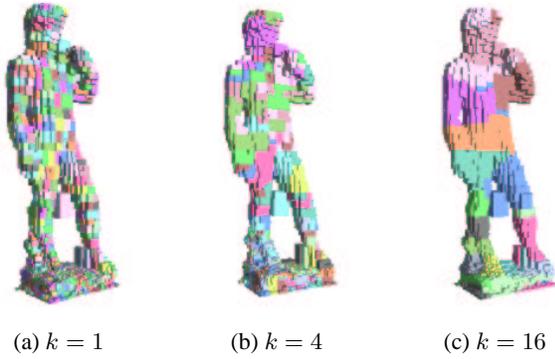


Figure 1: K-way replication enables sort-last pixel composition bandwidths closer to n-way replication ( $n = 16$ ) with memory capacity closer to 1-way replication. Color of each bounding box indicates which server renders its enclosed triangles. (Also shown in Color plate 1)

Unfortunately, this purely view-dependent partitioning approach requires the entire scene to be replicated on every processor.

Our new k-way replication approach avoids full replication of the scene data. But, it can employ view-dependent load balancing algorithms, since every primitive is available on more than one processor. With k-way replication, we expect to achieve performance similar to n-way replication, but with storage costs closer to 1-way replication (see Figure 1(b)).

## 4 System Overview

Our system executes in five stages, as shown in Figure 2. The first two stages are performed once per 3D model as preprocessing steps:

- Preprocess: Scene Graph Construction:** During the first preprocessing stage, the system takes as input an arbitrary 3D polygonal model and automatically builds a scene graph with multiple “levels of detail” [6]. The result is a set of *objects* arranged in a bounding volume hierarchy, in which each object stores a graphical representation with near-constant complexity (e.g., 500 polygons per object). The leaf objects store the original polygons of the input model (decomposed into spatially coherent groups), while the interior objects store simplified versions of their children. Details are provided in Section 5.1.
- Preprocess: Object Replication:** During the second preprocessing stage, the system assigns every object to be stored on  $k$  out of  $n$  server PCs. The goal of this stage is to distribute copies of the objects in a manner that allows a later load balancing algorithm to perform an effective view-dependent partition, while avoiding starvation of any server as the user zooms in to view a subset of the scene (as long as there is sufficient work to keep all servers busy). Details are provided in Section 5.2.

Once the preprocessing steps are completed, a user may view the model interactively with our system. The following three stages are executed for every frame of an interactive session. They are arranged in a three-stage asynchronous pipeline comprising one *client* PC,  $n$  server PCs, and a display PC connected by a system area network (see Figure 2):

- Client: Traversal and Assignment:** During the first on-line stage, the client PC traverses the multiresolution scene graph from top to bottom to determine which objects are visible from the user’s viewpoint and which have appropriate levels of detail to be rendered in an interactive frame time. It assigns each such object to be rendered on exactly one server. Then, it partitions the screen into a rectangular uniform grid of *tiles* and assigns each of them to be composited by a server. Details are provided in Sections 5.3-5.5.
- Server: Rendering and Pixel Composition:** In the next stage, every server PC renders all objects it has been assigned by the client, reads back the color and depth values of the resulting pixels, compresses them, and sends them over the network to the other servers compositing the overlapped tiles. Then, as the server receives pixels rendered by other servers, it decompresses them and composites them with depth comparisons into its local copy of the corresponding tiles, and sends the composited color values for each completed tile to a *display* PC. Details are provided in Sections 5.6-5.7.
- Display:** In the final stage, the display PC receives tiles of pixels from all the servers, and loads them into its frame buffer (without depth comparisons) for display. Details are provided in Section 5.8.

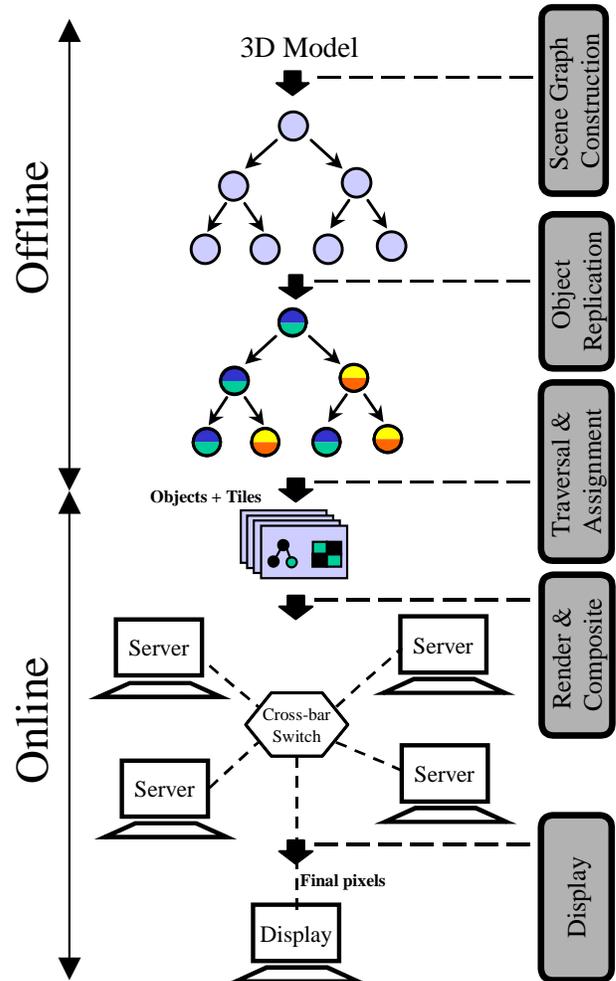


Figure 2: System organization.

## 5 Implementation

In this section, we investigate issues in building our prototype system and discuss algorithms for making each stage work effectively.

### 5.1 Preprocess: Scene Graph Construction

The first preprocessing stage of our system takes as input a static 3D model consisting of an unorganized set of polygons and automatically builds a multiresolution scene graph data structure from them. The goals are: (1) to create a coarse-grained index for the primitives to facilitate assignment to servers, (2) to construct a compact bounding volume hierarchy to facilitate view frustum culling, and (3) to augment the hierarchy with “levels of detail” (LODs) to enable the client to maintain a constant frame rate during later interactive viewing sessions.

Multiresolution scene representations have been well-studied, dating back to Clark’s seminal paper in 1976 [6]. In our system, we are motivated to choose a hierarchical structure of discrete objects, each comprising multiple polygons [6, 10]. This choice facilitates coarse-grained assignment of work to servers, which is important in a distributed system with high-latency communication; it enables time-critical selection of LODs suitable for interactive display; it allows use of retained display lists, which provide faster rendering rates on most graphics hardware; and, it enables relatively accurate estimates of rendering time due to aggregation of estimation errors over multiple polygons. In exchange for these benefits, we accept the sudden “pops” that come with discrete LOD switches, noting that polygons are smaller in our system and pops are less noticeable than in comparable single processor systems.

We have chosen to implement a version of Clark’s original scene graph structure [6] in which each object in the hierarchy contains a list of polygons representing a simplified version of its children. Our method for constructing the multiresolution hierarchy proceeds as follows (see Figure 3). Starting with the original model, we apply a sequence of simplification steps using Rossignac’s vertex merging algorithm [23] with greater and greater decimation factors. This produces polygonal representations for a discrete set of *levels*,  $L_i$ , usually with a little less than half as many polygons in each coarser level. We then partition the polygons for each level into a set of spatially compact objects using a k-d tree. Starting with the polygons of the coarsest level,  $L_0$ , binary splits are applied recursively along the dimension of the longest axis of its bounding box. The position of the split is chosen such that the two smaller objects have nearly equal numbers of primitives. The recursive splits continue until the number of polygons left in every object is below some threshold. A spatially coherent hierarchy is constructed by pre-applying the same splits made in level  $i$  to all finer levels  $j$  ( $j > i$ ) and by assigning each object of level  $i + 1$  as a child of the object at level  $i$  corresponding to same k-d cell (see Figure 3). The result is a bounding volume hierarchy of objects in which every object stores a polygonal representation of its descendants with approximately the same rendering complexity (e.g., 500 polygons per object).

Note that this structure expands the storage required for the model by  $\sum_{i=0}^d 1/b^i$ , where  $d$  and  $b$  are the depth and branching factor of the scene graph hierarchy, respectively. We usually choose  $b \sim 2$  in order to avoid too much storage overhead, while minimizing the effects of switching levels of detail.

### 5.2 Preprocess: Object Replication

The second preprocessing stage determines how to replicate objects on the servers. The goal is to develop a strategy that provides the on-line load balancing algorithm with sufficient choices for rendering assignments so that it can balance the load among the servers while avoiding starvation and large pixel composition overheads.

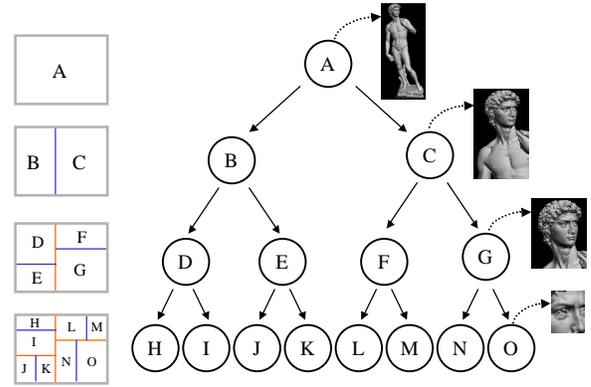


Figure 3: Construction of the multiresolution scene graph. The scene is first simplified successively (coarser levels are higher in the tree). Then, each level is decomposed into objects containing nearly the same number of polygons with recursive binary space partitions (dark lines). This structure allows the client to perform efficient view frustum culling and select an appropriate level of detail to maintain an interactive frame rate.

Achieving this goal is non-trivial due to the unpredictable access patterns of a multiresolution scene graph traversal. Since each object is rendered only for certain viewpoints, and every server stores only a partial subset of the objects, there is a significant risk that servers might be starved of work when the user zooms in to view a subset of the model. This concern implies that primitives replicated on each server should be uniformly distributed throughout the scene graph in order to avoid starvation (see Figure 4(a)).

On the other hand, less overheads due to pixel composition can be achieved if objects replicated on each server are clustered in screen-space. This observation implies that objects resident on each server should be arranged in large, spatially coherent groups (e.g., “cubes” or “slabs” [15, 21]) so that the cumulative screen-space overlaps of objects rendered by different servers is small. However, in this case, if the user zooms in to view a smaller portion of the data, some servers may be starved for work, which would cause imbalances in the rendering loads (see Figure 4(d)).

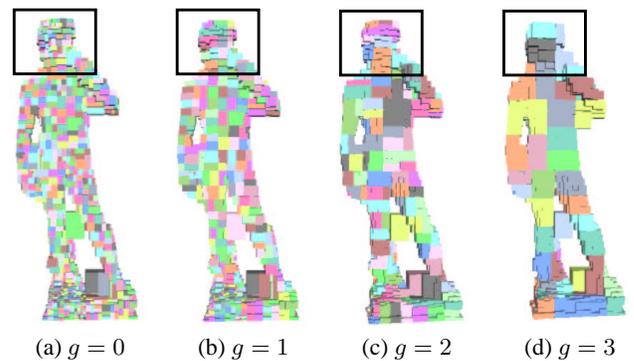


Figure 4: Visualization of varying  $g$ . The overlaid rectangle indicates a frame in which the user zooms in on the head. The color of each bounding box indicates on which server the enclosed triangles are stored. (Also shown in Color plate 2)

Our solution is to store on each server a set of objects arranged in spatially compact “cubes” at  $k$  different granularities. Since every object can be replicated  $k$  times, it can be grouped with other objects in  $k$  cubes of different sizes suitable for different

viewpoints. The finest granularity corresponds to distributing objects among servers in a round-robin fashion – which is suitable for avoiding starvation when the viewpoint is zoomed close to the object. Coarser granularities correspond to sets of objects on the same server residing in larger “cubes” – which are suitable for more zoomed out viewpoints. The motivation for this approach is to allow a later on-line, view-dependent load balancing algorithm to be able to choose a set of spatially co-located objects for each server at the granularity that both avoids starvation and minimizes pixel composition overheads.

In our implementation, we use the scene graph structure to find spatially co-located groups of objects with different granularities. Conceptually, “cubes” are constructed by replicating groups of objects that are close to each other in the bounding volume hierarchy on the same server. We use a “granularity” parameter  $g$  to indicate that objects should be replicated on a server by themselves ( $g = 0$ ), with their siblings ( $g = 1$ ), with their siblings and first cousins ( $g = 2$ ), and so on. When  $k$  is greater than 1, the resulting tree is equivalent to combining trees with multiple granularity parameters.

Figure 5 shows schematic replication results for a simple scene graph using our  $k$ -way approach. The shade(s) shown in each object, indicate the server(s) on which it is replicated. The top row (Figure 5(a-c)) shows examples for  $k=1$  (each object has a single shade). Note that when  $g = 0$ , each object is assigned to a server distinct from its siblings. When  $g = 1$ , if we look at any given object, then we notice that its siblings are assigned to the same server. At  $g = 2$ , this condition extends to include cousins as well. Since child nodes are spatially co-located, this strategy allows us to form “blocks” of objects. In Figure 5(d-f), we set  $k = 2$  and hence each object now has two distinct shades. In this case, the replication patterns are a combination of the ones for  $k = 1$ . For instance, the pattern in the lower half of (e) ( $k = 2, g = 1$ ) is equivalent to the one in (b) ( $k = 1, g = 1$ ), and the pattern in the upper half is equivalent to the one in (a) ( $k = 1, g = 0$ ).

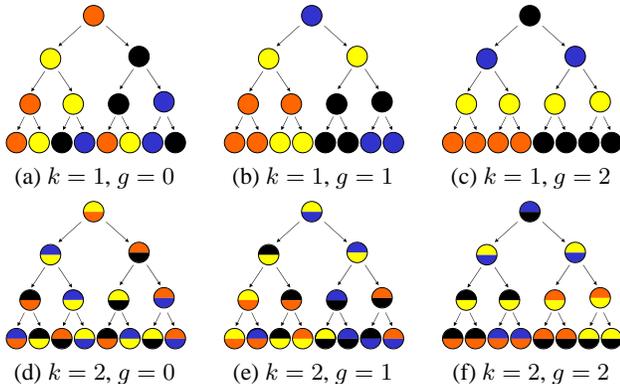


Figure 5: Schematic diagram showing how every object (circle) is replicated with the *multiresolution cubes* algorithm. The different granularities cause trade-offs between screen-space coherence and starvation at run-time.

### 5.3 Client: Scene Graph Traversal

Once the preprocessing steps are complete, a user can view the model under interactive control of a client machine. It loads the multiresolution scene graph (without the polygons) and traverses the scene graph to find a set of objects for each server to render in every frame. During the traversal, it culls objects to the current view-frustum using hierarchical intersection tests with their bounding spheres. A unique feature of our system is that the client integrates load balancing into a multiresolution scene graph traversal.

In this way, it is able to predict the rendering times and overheads for each server and use the estimates to guarantee an interactive frame rate.

For each frame, the scene graph traversal proceeds by visiting objects from top to bottom. Initially, the root object is assigned to a server and placed on a queue. Then, as each object is popped off the queue, it is marked “unassigned,” and all its children whose bounding spheres intersect the view frustum are added to the queue and assigned to servers. The process continues until the predicted processing time of the maximally loaded server reaches a user-specified time bound, or until the pixels/polygon density of objects not yet on the queue is below some threshold.

This multiresolution assignment procedure allows our system to either (1) vary the scene detail to maintain a target frame rate, or (2) vary the frame rate to maintain a specified level of detail. We believe that this is the first system to integrate time-critical rendering with coarse-grained parallelism.

### 5.4 Client: Object Assignment

During the multiresolution scene graph traversal, the client assigns each object to be rendered by a server. The goal is to assign objects in a manner that balances the load, minimizes overheads, and adheres to the assignment constraints imposed by  $k$ -way replication.

As with any parallel system, the client must trade-off two factors when assigning objects: balancing the work loads and minimizing the overheads. In our system (for small  $k$ ) the former factor is largely linked to starvation, while the latter factor is linked to avoiding pixel composition – i.e., minimizing the number of screen-space tiles overlapped by objects assigned to different servers.

Of course, finding an optimal solution for this assignment problem is NP-hard, and thus we focus our search on effective heuristics. We have experimented with two greedy strategies. The first one assigns objects to the least loaded server (“Least Server First”), focusing only on load balance. The second strategy is motivated by reducing pixel composition overheads. It assigns objects to the server whose estimated pixel redistribution overheads will increase the least. Yet, if the work load previously assigned to that server is already above a per server threshold, it searches the list of objects that were previously assigned to it and moves to another server the object whose incremental overheads increase the least. This last step helps balance the load and avoid starvation.

Within the second strategy, we have tried two methods. One explicitly computes incremental object tile overlaps by “rasterizing” the object’s bounding box into the tile grid and counting the number of new tiles that must be read for each server if the object is assigned. We call this method “Least Cost First”. The other method uses a simpler computation: each server is represented by a “dot” on the screen, computed as the average of the screen-space projections of the centroids of objects assigned to the server in the previous frame. The relative cost of assigning an object to a server is approximated by the distance between the server’s dot and the projection of the object’s centroid. We call this method “Closest Dot First”. The intuitive motivation for this method is that it can utilize global knowledge about the distribution of objects from the previous frame, and thus servers can “map out” separate regions of the screen utilizing frame-to-frame coherence, thereby reducing pixel composition overheads.

### 5.5 Client: Tile Assignment

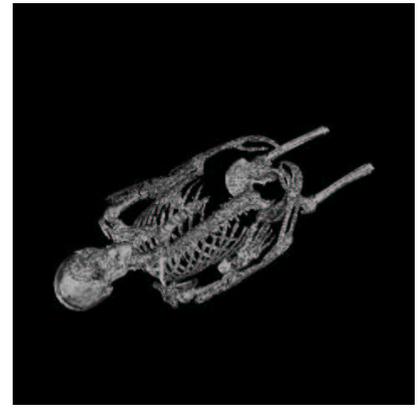
After all objects have been assigned to servers, the client partitions the screen into tiles and assigns each server a set of tiles to composite. For each tile assigned to a server, other servers will send their rendered pixels for the tile so that they can be composited with depth comparisons to form a complete image for the tile. The



(a) Michelangelo's "David" [13]  
(8,254,150 triangles, ~800MB)



(b) Face of Michelangelo's "St. Matthew" [13]  
(6,755,412 triangles, ~700MB)



(c) Visible Man Skeleton (without feet) [14, 28]  
(2,432,525 triangles, ~250MB)

Figure 6: Test models used in our experiments.

goal of the client in this stage is to distribute the pixel composition work evenly among the servers, possibly filling in any imbalances created by uneven object assignments.

We have implemented two algorithms for the tile assignment. The first algorithm assigns the tiles in a static interleaved manner. This approach suffers from imbalances, as the work to composite some tiles is much more than others. The second arranges the tiles in decreasing order of predicted work and then performs a greedy first-fit-decreasing assignment. This second algorithm consistently outperforms static interleaved assignment, and thus we use it for all the results presented in later sections.

## 5.6 Server: Rendering

Once the client has partitioned the work, the servers render their respective portions of the model. In our system, the servers store the geometry corresponding to its subset of the objects as a vertex array. We use the `glInterleavedArrays()` function in OpenGL to render the arrays with vertex normals and lighting. We currently measure approximately 2.4M triangles per second with this method using nVidia GeForce II graphics cards. We note that triangle strips could generally be used in order to further improve rendering performance on each server. However, finding the fastest rendering mode for each API and graphics card is outside the scope of this study.

## 5.7 Server: Peer-to-Peer Pixel Composition

After objects are rendered, the servers are responsible for depth compositing the resulting pixels to form the final image. The composite operations are performed by the servers in a peer-to-peer fashion. The client sends information to servers indicating which server is responsible for a given screen-space tile. Servers that have rendered to tiles for which they are not responsible for the compositing operations must read back these tiles from the framebuffer and forward these (with both color and depth information) to the server in charge of this tile.

Two interesting issues arise during this stage. First, reducing the total amount of data transferred over the network is important, since network bandwidth is a potential bottleneck in our system. Second, the servers must transmit data in a manner such that the network is utilized efficiently by reducing contention and hiding transfer latencies.

To reduce the required network bandwidth, the tile images are compressed with run-length encoding after they are read back from

the framebuffer. In our tests we notice that we are able to cut down network traffic by 50% as a result of this straightforward and efficient compression scheme. An added advantage of this scheme is that we are able to perform the depth composite of a tile using its compressed representation. This reduces the number of compare operations required leading to faster depth compositing of tiles.

As network contention is a critical issue in PC clusters, scheduling network transfers carefully is important. One option is to have each server render all of the objects assigned to it and then send the pixels of its rendered tiles all at once. This creates bursty traffic on the network. Instead, we choose to interleave rendering and network sends. When the client informs the server which objects should be rendered, it indicates which objects overlap each tile. This allows the server to render all the objects that overlap a tile, read back the color and depth for the resulting pixels, and send them to the appropriate server before proceeding to the next tile. This communication pattern utilizes the network far more efficiently and we notice significantly lower latencies in our system.

After the server has received all pixels for a tile from its peers and composited them together, it sends the resulting color pixel values to the display.

## 5.8 Display

The display receives tiles that have been composited by the servers and copies them into its framebuffer (without depth comparisons). The resulting images are projected on a rear projection display.

# 6 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 our design and algorithms. The goals of the experiments are to answer the following questions:

- Which object assignment algorithms work best?
- What is the effect of varying the replication factor ( $k$ )?
- What is the effect of varying the replication granularity ( $g$ )?
- Can the system maintain a constant frame rate?
- Does our system perform well?

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.

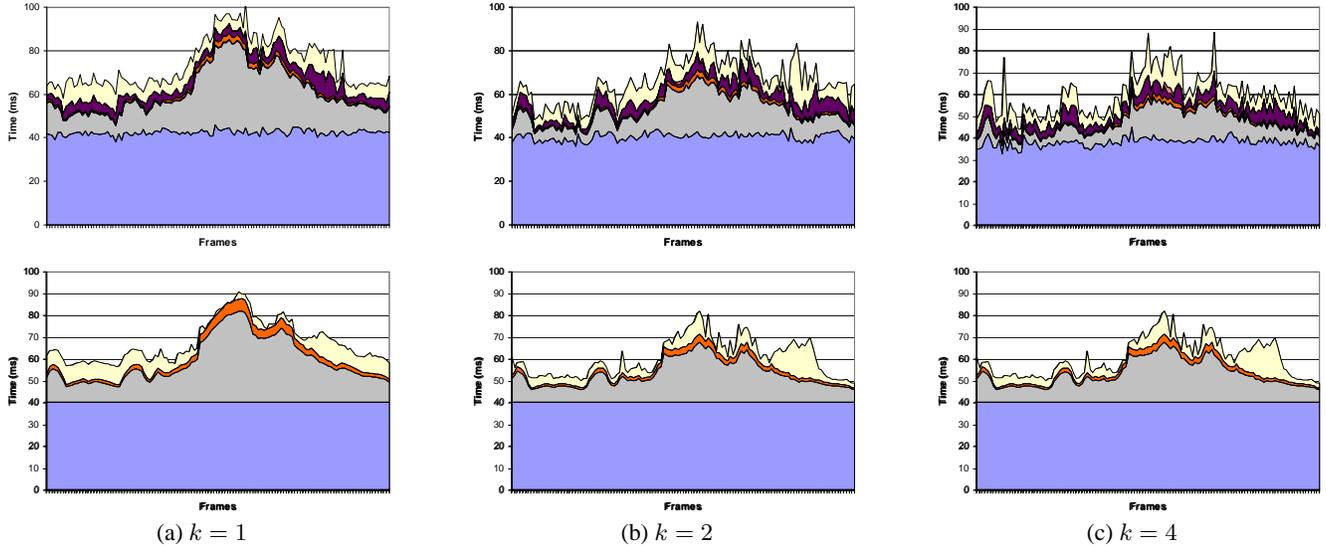


Figure 7: Comparison of measured (top) and simulated (bottom) results for our tests with the David model using 16 server PCs. The times are decomposed into sections representing (from bottom to top) render time (blue), pixel read time (gray), pixel write time (orange), network wait time (red), and imbalance time (yellow).

## 6.1 Experimental Platform

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 [3]. 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 cross-bar switches. We have used the GM driver for Windows 2000 provided by Myricom. The total cost of the system is around \$50K.

Experiments were executed with the three test models shown in Figure 6. 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.

## 6.2 Comparison of Simulated and Measured Data

We have implemented both a real working system and a simulation. The working system provides our main results. However, it is sometimes interesting to use simulations to investigate the predicted performance of our approach in cases impractical to run on the real system. For instance, the Michaelangelo sculptures do not fit fully in memory ( $k = n$ ) on our server PCs. In experiments where these data points are relevant, we present simulation results. We present real system results in the final subsection titled “Performance Results.”

To validate our simulation results, we show both real and simulation results for  $n = 16$  and  $k = 1, 2, 4$  in Figure 7. The total

height of each plot represents the time in milliseconds required for the slowest server in each frame of the test. The imbalance time represents the difference between the slowest server and the average one. From these plots, which are representative of others, our simulation data is quite close to our real measurements.

## 6.3 Evaluation of Object Assignment Algorithms

In our first experiment, we evaluated the performance of our system with different object assignment algorithms using the David model with  $n = 16$  and  $g = 0$ . The goal of the experiment is to determine which algorithms work best and which factors dominate the system performance. Because we were interested in the performance of each algorithm when  $k = n$ , we present simulation data here.

The results are shown in Table 1. The first column lists the algorithm, second is the replication factor, the next four list the average render, pixel read, pixel write, and imbalance times (same as in the plots). The sum of these four times is the Frame Time. Finally, the efficiency (render time / frame time) is listed in the rightmost column.

Alg	$k$	Render Time	Read Time	Write Time	Imbal Time	Frame Time	Effic (%)
Least Server	2	40.03	18.60	2.57	0.27	61.46	67.21
	4	40.03	18.15	2.51	0.35	61.03	67.60
	8	40.03	18.94	2.61	0.18	61.75	66.83
	16	40.03	19.34	2.67	0.17	62.21	66.43
Least Cost	2	40.03	15.29	2.11	4.21	61.64	66.60
	4	40.03	11.87	1.64	6.51	60.04	68.16
	8	40.03	9.70	1.33	13.36	64.41	63.12
	16	40.03	4.72	0.62	3.89	49.26	82.28
Closest Dot	2	40.03	13.25	1.83	5.25	60.35	67.57
	4	40.03	9.56	1.31	6.43	57.33	71.26
	8	40.03	8.27	1.14	5.97	55.40	74.02
	16	40.03	4.33	0.57	4.42	49.35	82.05

Table 1: Comparison of object assignment algorithms during simulation with David model for  $n = 16$ . All times are in milliseconds.

As expected, the “Least Server” algorithm provides very good balance (the imbalance times for all  $k$  are at most .35ms). But, it incurs more pixel read and pixel write overheads ( $>18$ ms) than the other algorithms, and thus its efficiencies are less for most values of

$k$ . Somewhat unexpectedly, the “Closest Dot” algorithm performs better than “Least Cost” in all cases, except when  $k = n$ . It seems that more exact local cost estimates for the current frame are not as useful as global information garnered from the previous frame.

The client times average between 20ms and 35ms. Since this is a single stage in a pipelined system, we notice that the client time is unimportant as the servers are our bottleneck. We find that in our current system, the client is fast enough to drive the servers at interactive frame rates.

#### 6.4 Analysis of Replication Factor and Granularity

We next conducted an experiment to evaluate the trends of performance by varying the replication factor ( $k$ ) and granularity ( $g$ ) during simulations with the David model running on 16 servers. We used the “Closest Dot” algorithm. Our hypothesis is that even small replication factors will provide performance comparable to full  $n$ -way replication.

Table 2 shows the simulation results with values of  $g$  ranging from fine granularity ( $g = 0$ ) to coarse granularity ( $g = 3$ ), and  $k$  ranging from no replication ( $k = 1$ ) to full replication ( $k = 16$ ). The meaning of each column is the same as Table 1, except for the one labeled “Avail Ratio,” which is the minimum number of visible primitives available to each server divided by the average number primitives rendered by a server each frame. A value less than 1.0 means that a server is definitely starved for rendering work. Two main trends are noticeable in this table.

$g$	$k$	Render Time	Read Time	Write Time	Imbal Time	Frame Time	Avail Ratio	Effic (%)
0	1	40.0	19.1	2.6	5.8	67.5	0.9	60.3
	2	40.0	13.3	1.8	5.3	60.4	1.6	67.6
	4	40.0	9.6	1.3	6.4	57.3	3.3	71.3
	8	40.0	8.3	1.1	6.0	55.4	3.5	74.0
	16	40.0	4.3	0.6	4.4	49.4	16.0	82.1
1	1	40.0	16.2	2.2	10.0	68.4	0.7	59.8
	2	40.0	12.1	1.7	9.3	63.1	1.5	64.2
	4	40.0	9.9	1.4	6.1	57.4	2.4	71.1
	8	40.0	7.7	1.1	4.7	53.5	5.2	75.9
	16	40.0	4.3	0.6	4.4	49.4	16.0	82.1
2	1	40.0	12.1	1.7	20.7	74.5	0.6	55.3
	2	40.0	10.8	1.5	10.7	63.0	1.2	64.9
	4	40.0	9.3	1.3	14.7	65.2	1.8	62.2
	8	40.0	8.0	1.1	4.9	54.0	4.1	75.5
	16	40.0	4.3	0.6	4.4	49.4	16.0	82.1
3	1	40.0	8.7	1.2	30.5	80.4	0.4	51.0
	2	40.0	8.7	1.2	25.0	74.9	0.6	54.6
	4	40.0	7.7	1.1	28.2	77.05	0.8	53.4
	8	40.0	7.8	1.1	4.8	53.7	4.9	75.8
	16	40.0	4.3	0.6	4.4	49.4	16.0	82.1

Table 2: Comparison of performance for different replication factors ( $k$ ) and granularities ( $g$ ) for simulations with the David model for  $n = 16$ . All times are in milliseconds.

First, as the granularity ( $g$ ) of objects assigned to the same server increases, we see that the pixel read and write times decrease, but the imbalance times increase, causing overall efficiencies to vary only slightly. The reason is because higher values of  $g$  represent larger 3D clusters of objects replicated together on the same server. This enables the view-dependent load balancing algorithm to assign objects to servers with less cumulative screen overlaps when the viewpoint is “zoomed out”. Yet, it causes servers to become starved (Avail Ratio is low) when the replication factor is small (e.g.,  $k \leq 4$ ) and the viewpoint is “zoomed in” (see Figure 4(d)). Overall, we find that fine to moderate granularity (of replication  $g = 0$  or  $g = 1$ ) provides the best results on average in our tests.

Second, as the replication factor ( $k$ ) increases, we see that the system’s efficiency increases. The reason is due to reduced pixel read and write times. With larger  $k$ , the view-dependent load balancing algorithm has fewer constraints regarding which objects can

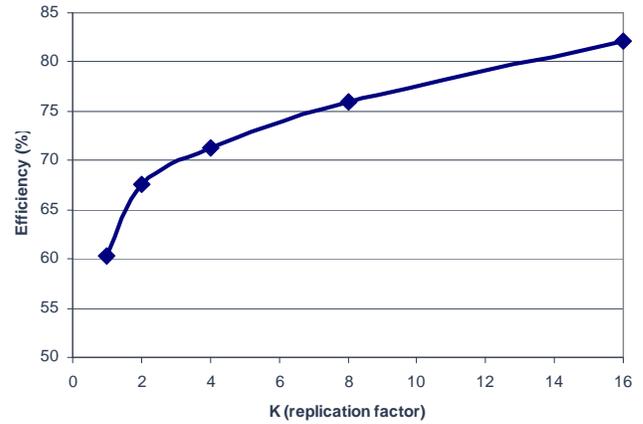


Figure 8: Plot of efficiency as we vary replication factor ( $k$ ). Note that the Y-axis begins at 50%.

be assigned to each server, and thus it can partition the objects into sets with greater screen locality (see Figure 1). The net result is less pixels to be rendered, read back from the frame buffer, transmitted over the network, and composited by the servers. We note that the efficiency improvement is non-linear (see Figure 8). This is a positive result. It confirms that the  $k$ -way approach is an effective way to achieve view-dependent object assignments with small replication factors.

#### 6.5 Maintaining a Uniform Frame Rate

Our next experiment compares the results of our system rendering a fixed number of polygons versus maintaining a uniform frame time. The difference is whether the overheads due to pixel reads and transfers are taken into account by the multiresolution assignment algorithm. The graphs are shown in Figure 9. We observe that the algorithm that maintains a fixed number of polygons has a uniform render time. However, the overheads due to pixel transfers and compositing cause the final frame time to vary significantly. In contrast, we see that the time critical algorithm is able to take the overheads into account during the assignment phase, allowing it to trade-off render time (by reducing detail) for overhead time in order to keep the sum nearly constant. This result indicates that it is indeed feasible to combine multiresolution techniques for maintaining uniform frame rates with a parallel rendering algorithm, taking both rendering time and parallelization overheads into account.

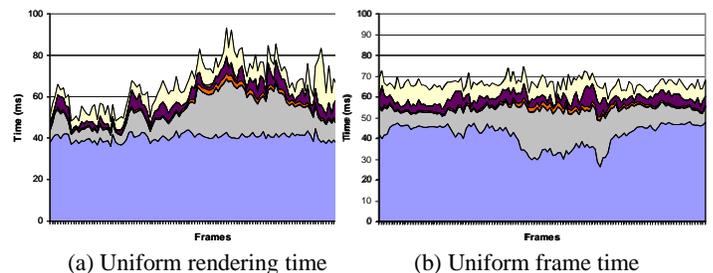


Figure 9: Comparison of assignment algorithm with uniform rendering time and with uniform frame time (including overheads). The times are decomposed into sections representing (from bottom to top) render time (blue), pixel read time (gray), pixel write time (orange), network wait time (red), and imbalance time (yellow).

## 6.6 Performance Results

In our final experiment, we tested the performance achievable with our system using  $k$ -way replication for all three test models. For simplicity of comparison, we set the total number of polygons rendered by all server PCs in each frame to be 2.4 million in every test. Note that this translates to around one frame per second on a single PC.

An interesting aspect of this experiment is that we can load more copies of the data as more PCs (and more memory) are added to the system. Essentially, the  $1/n$  portion of the model for each server gets smaller as  $n$  grows, leaving room in memory for more replicated copies of the data (greater  $k$ ). In these experiments, we set  $[k = \frac{1}{4}n]$  for  $n = 2, 4, 8, 16, 24$ . We used the “Closest Dot” algorithm and  $g = 1$ . To obtain results for  $n = 2$ , we temporarily added extra memory to the two server PCs.

Figure 10 shows 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 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.

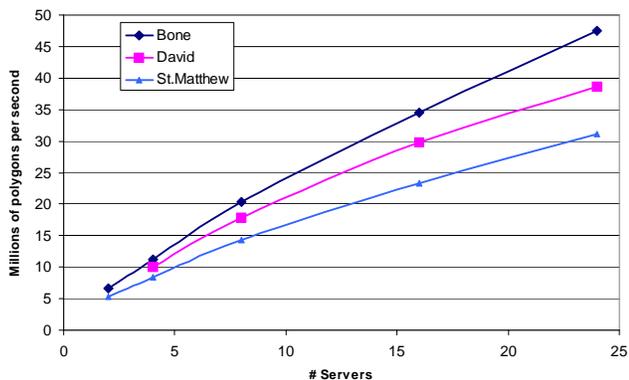


Figure 10: Plot of millions of polygons per second rendered by our real system during tests with three large models.

## 7 Discussion

Our study has shown that using a multi-stage hybrid parallel rendering pipeline with  $k$ -way replication works effectively on a PC cluster to render a large, detailed scene database that does not fit in the memory of a single PC. Yet, many problems remain open for future research and investigations.

First, the sort-last based parallel rendering pipeline described in this paper does not enforce primitive ordering. While this is not a significant limitation for the visualization applications we have tested, it would be a problem for ones that rely upon transparency. To address this problem, we could partition objects into layers to be composited in back-to-front order [22]. This approach is an interesting topic for further study.

Second, the approach described in this paper only extends the size of models by  $n/k$  where  $n$  is the number of rendering servers and  $k$  is the replication factor. We have shown that  $k = 4$  works quite well. However, data visualization applications may require an out-of-core method to store and traverse their databases. Previous studies have focused on out-of-core techniques for a single

machine [5, 8]. We believe that  $k$ -way replication principles can apply to out-of-core database management for multiple computers. For instance, it may be beneficial to store multiple copies of each data element on different disks in order to provide flexibility regarding which computers load and process data. This is an interesting and important area of future research.

Third, the study reported in this paper considers a display at 1Kx1K resolution, and the current system area network bandwidth is not adequate for software peer-to-peer image composition for displays with much higher resolution. It would be interesting to investigate how to use the  $k$ -way replication approach with display walls and other distributed frame buffers [2, 11]. Samanta et al. [26] proposed a screen partitioning approach for a PC cluster that drives a multi-projector display system, but their method requires a copy of the database on every rendering server. It is worthwhile to study how to integrate their methods with our  $k$ -way approach.

Fourth, our current system has a single client. Certain parallel rendering systems such as WireGL [4] separate clients from rendering servers and allow multiple clients to drive the rendering servers. We believe that the  $k$ -way replication and assignment algorithms proposed in this paper can apply to how a scene database is stored on multiple clients.

Finally, we have only worked with static scenes in this study. Supporting dynamic scenes would require updates to the scene graph during the on-line stages of our process. An extension for dynamic rigid body motions seems possible and presents an interesting topic for further work. Similarly, one could consider integrating  $k$ -way replication as a caching mechanism for a system with dynamic data migration.

## 8 Conclusion

This paper describes the design, implementation and experiments of a multi-stage hybrid parallel rendering system for a PC cluster. Our work makes two main contributions.

First, we have proposed and demonstrated a  $k$ -way replication approach to distribute a large scene database across multiple PCs. This approach takes advantage of the aggregate memory resources in a PC cluster (in addition to the aggregate rendering power) to render a detailed scene database far larger than the memory of a single PC. We have demonstrated that our rendering pipeline from client to display can render a complex scene database such as Michelangelo’s David at the rate of about 40 million polygons a second on a 24 rendering server cluster at an interactive frame rate.

Second, we investigated replication strategies and load balancing algorithms within the framework of our  $k$ -way replication approach. In our experimental studies, we showed that a simple method for dynamic screen partitioning based on proximity to dots representing servers performs quite well. We learned from our experiments that with a small  $k$ , the efficiency improves quickly, approaching the  $n$ -way (full) replication case. For example, we have showed that  $k = 4$  works quite well for medium size clusters such as 24 rendering servers. Choosing  $k$  is a way to trade off communication and memory usage.

This is a first step towards studying how to design a parallel graphics pipeline to trade off communication bandwidth and aggregate memory resources for a PC cluster.

## References

- [1] James Ahrens and James Painter. Efficient sort-last rendering using compression-based image compositing. *Second Eurographics Workshop on Parallel Graphics and Visualisation*, September 1998.
- [2] W. Blanke, C. Bajaj, D. Fussell, and X. Zhang. The metabuffer: A scalable multiresolution multidisplay 3-d graphics system using commodity rendering engines. Technical Report TR2000-16, University of Texas at Austin, 2000.
- [3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE MICRO*, 15(1):29–36, February 1995.
- [4] Ian Buck, Greg Humphreys, and Pat Hanrahan. Tracking graphics state for networked rendering. In *Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 87–98. ACM Press, August 2000.
- [5] Yi-Jen Chiang, Claudio T. Silva, and William J. Schroeder. Interactive out-of-core isosurface extraction. *IEEE Visualization '98*, pages 167–174, 1998.
- [6] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.
- [7] Michael Cox. *Algorithms for Parallel Rendering*. PhD thesis, Department of Computer Science, Princeton University, 1995.
- [8] Michael Cox and Narendra Bhandari. Architectural implications of hardware-accelerated bucket rendering on the pc. In *Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 25–34, Los Angeles, CA, 1997.
- [9] Michael Cox and Pat Hanrahan. Pixel merging for object-parallel rendering: a distributed snooping algorithm. In Thomas Crockett, Charles Hansen, and Scott Whitman, editors, *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 49–56. ACM, November 1993.
- [10] Carl Erikson, Dinesh Manocha, and William V. Baxter III. Hlods for faster display of large static and dynamic environments. *2001 ACM Symposium on Interactive 3D Graphics*, pages 111–120, March 2001.
- [11] Alan Heirich and Laurent Moll. Scalable distributed visualization using off-the-shelf components. In *IEEE Parallel Visualization and Graphics Symposium*, pages 55–59, October 1999.
- [12] Greg Humphreys, Mathew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: A scalable graphics system for clusters. In *To appear, Computer Graphics (SIGGRAPH 2001)*, 2001.
- [13] 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, *Computer Graphics (SIGGRAPH 2000)*, pages 131–144, 2000.
- [14] William Lorenson. Marching through the visible human. In *The Visible Human Project Conference Proceedings*, October 1996.
- [15] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.
- [16] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed rendering using image composition. In *Computer Graphics (SIGGRAPH 92)*, pages 231–240, 1992.
- [17] 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.
- [18] Carl Mueller. The sort-first rendering architecture for high-performance graphics. In *2001 ACM Symposium on Interactive 3D Graphics*, March 1995.
- [19] Carl Mueller. Hierarchical graphics databases in sort-first. In *Proceedings of the IEEE Symposium on Parallel rendering*, pages 49–57, 1997.
- [20] Ulrich Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 97–104. ACM, November 1993.
- [21] Ulrich Neumann. Volume reconstruction and parallel rendering algorithms: A comparative analysis. Ph.d. thesis, University of North Carolina at Chapel Hill, 1993.
- [22] Thu D. Nguyen and John Zahorjan. Image layer decomposition for distributed rendering on commodity clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2000.
- [23] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. *Geometric Modeling in Computer Graphics and Applications, Springer-Verlag*, pages 455–465, 1993.
- [24] 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.
- [25] 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.
- [26] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Load balancing for multi-projector rendering systems. In *Eurographics/SIGGRAPH workshop on Graphics hardware*, pages 107–116, Los Angeles, CA, August 1999. ACM Press.
- [27] Sony. Gscube. *SIGGRAPH 2000 Tradeshow*, 2000.
- [28] Greg Turk. Large geometric models archive. [www.cc.gatech.edu](http://www.cc.gatech.edu), 2000.
- [29] T. Udeshi and C. Hansen. Parallel multipipe rendering for very large isosurface visualization, 1999.
- [30] Bin Wei, Douglas W. Clark, Edward W. Felten, Kai Li, and Gordon Stoll. Performance issues of a distributed frame buffer on a multicomputer. In Stephen N. Spencer, editor, *Proceedings of the Eurographics / Siggraph Workshop on Graphics Hardware (EUROGRAPHICS-98)*, pages 87–96, New York, August 31–September 1 1998. ACM Press.