DDDDRRaW*: A Prototype Toolkit for Distributed Real-Time Rendering on Commodity Clusters

Thu D. Nguyen, Christopher Peery, and John Zahorjan†
{tdguyen, peery}@cs.rutgers.edu, zahorjan@cs.washington.edu

Technical Report DCS-TR-420
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

October 1, 2000
Revised December 14, 2000

Appears as a long paper in Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), May 2001

Abstract

We describe DDDDRRaW, a prototype toolkit for real-time rendering on clusters of commodity computers. In contrast to most work on cluster computing, DDDDRRaW supports a repeated, low-latency computation, the drawing of frames, which must take place on a time scale of 30-100 ms. DDDDRRaW employs *Image Layer Decomposition*, a rendering-specific work partitioning algorithm described and evaluated via simulation in [13]. In this paper, we address implementation issues not confronted in the design of ILD. In particular, one important issue that must be confronted is how to exploit the potential parallelism afforded by the multiple hardware resources of each node: the CPU, the network adapter, and the video card.

We evaluate DDDDRRaW's live performance on two small workstation clusters representing different points in the technology spectrum. Our results show that DDDDRRaW effectively exploits cluster resources to improve real-time rendering performance and should scale well to moderately sized clusters.

1 Introduction

We describe DDDDRRaW, a prototype distributed rendering toolkit developed specifically for clusters of commodity computers. This work is motivated by our desire to increase the *real-time* rendering capabilities available to users of *typical* workstations. Specifically, we propose to exploit clusters of workstations, where each node is equipped with a hardware graphics accelerator, to provide real-time rendering performance that is one to two generations ahead of what is achievable on a single machine. The steeply rising performance curve of hardware graphics accelerators has attracted an increasing variety of interactive 3D applications to such platforms. At the same time, there is a demand for virtually unbounded rendering performance because of the desire for increasingly more accurate representations of the real world [1].

While the commodity cluster is an attractive platform because of its ubiquity, low cost, and ease of expandability, it presents a number of challenges. In particular, a cluster-based distributed real-time renderer must be structured to: *(i)* use the multiple hardware graphics accelerators in the cluster to increase rendering performance over what is achievable by a sequential renderer that makes use of an accelerator, *(ii)* only impose overheads compatible with the 30-100 ms per-frame compute load of the application, *(iii)* minimize variance in the frame rate because variance is important to a user's perceived quality of service, and *(iv)* decouple communication bandwidth requirements from the complexity of the scene and the number of nodes used to achieve application and system scalability.

* DDDDRRaW is pronounced *draw* and stands for Distributed 3D Real-Time Rendering at Washington.
† Department of Computer Science & Engineering, University of Washington, Seattle, WA.

We designed and implemented DDDDRRaW specifically to meet these challenges. DDDDRRaW is implemented as a library that supports the distributed rendering of VRML scenes [20]. To assess DDDDRRaW's performance, we implemented a simple VRML viewer on top of DDDDRRaW. The viewer allows a user to interactively navigate through VRML 3D scenes using the mouse. Both the DDDDRRaW library and VRML viewer are based on the VRweb sequential VRML browser [15].

Like any system that distributes work over a cluster, workload partitioning is a central issue. DDDDRRaW employs *Image Layer Decomposition* (ILD), a scheme first presented in [13]. While ILD is not itself the topic of this paper, some of its characteristics have a bearing on the problems we had to confront in building DDDDRRaW and its resulting performance. For that reason, an overview of ILD is given first in Section 3 of this paper. (Section 2 discusses related work.) We then describe DDDDRRaW's architecture, an ILD optimization, and how rendering could be overlapped with communication. Finally, we evaluate DDDDRRaW's live performance using walkthroughs of 6 VRML scenes collected from the web.

Our results show that:

- DDDDRRaW effectively exploits cluster resources to increase real-time rendering performance on two small clusters despite several significant trade-offs of performance for implementation simplicity. On a cluster of 5 SGI O2s with hardware-accelerated rendering, DDDDRRaW speeds up the average frame rate by factors of 2.5-5.2 for 4 of the test scenes, 1.7-7.7 for all 6 scenes when communication is overlapped with computation. On a cluster of 8 PCs with software-only rendering, DDDDRRaW increases the average frame rate by factors of 2.3 - 5.7.
- A carefully implemented ILD-based distributed renderer should scale well to moderately-sized clusters (~16 nodes) comprised of existing commodity hardware components.
- The average frame rate can be improved significantly by overlapping communication with computation. This optimization requires sufficient hardware support, however. If both rendering and communication contend for the CPU, then overlapping communication and computation significantly increases frame latency. This implies that, unlike for parallel/distributed applications where job response time is the performance metric, this optimization is not always appropriate for real-time rendering.

2 Related Work

In this section, we discuss previous work on distributed polygon rendering. We focus on polygon renderers because polygon rendering is the most common technique supported by hardware accelerators and is the method employed in DDDDRRaW. The basic task in polygon rendering can be viewed as the sorting of scene primitives from their world coordinates to the screen [19]. To date, most polygon distributed renderers can be categorized into three classes, *sort-first*, *sort-middle*, and *sort-last*, depending on whether the sorting process takes place during the geometric transformation phase, between the geometric transformation phase and the rasterization phase, or after the rasterization phase [9].

In sort-first (e.g., [22, 12, 16]) and sort-middle (e.g., [4, 11]), the geometric transformation and rasterization of a polygon may be performed by different nodes, depending on the specific work assignment of each frame, typically requiring the redistribution of a significant number of primitives. This per-frame redistribution is a fundamental problem in our targeted environment because: *(i)* it requires either recomputing the geometric transformation of primitives that must be redistributed or accessing information that may be hidden inside a hardware graphics pipeline, and *(ii)* its bandwidth requirement directly depends on scene complexity. In contrast, DDDDRRaW leverages any available hardware acceleration and its bandwidth requirement is independent of scene complexity.

Sort-last (e.g., [10, 7]) corresponds to a data partitioning, where each node is assigned a subset of the polygons in the scene without any restrictions on the position of the polygons. Each frame, once each node has rendered its assigned polygons, the pixels must be sorted, typically using Z-buffering [5]. While compatible with our environment, sort-last is less than ideal because: *(i)* rendering nodes must send their Z-buffers along with the rendered pixels for the composition of the final image, which approximately doubles the required bandwidth, and *(ii)* primitives are typically assigned to renderers without regard to where they map to in screen space, implying that each renderer must typically send the entire image each frame. In contrast, DDDDRRaW does not require the communication of Z-buffer information across the network and is designed to minimize overlap between the images rendered on different nodes.

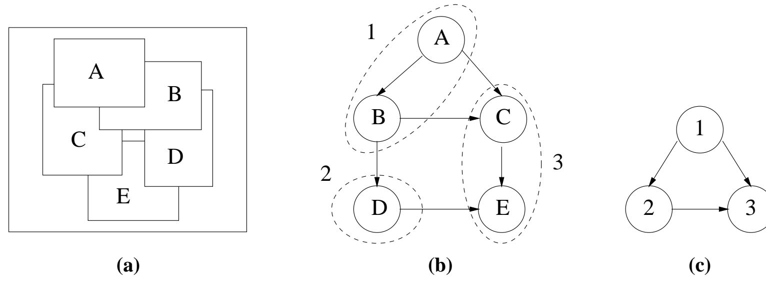

Figure 1: *(a) A particular view of a scene with 5 objects, (b) the corresponding occlusion graph and a partition of it into 3 subgraphs, and (c) the corresponding partition-occlusion graph.*

3 Image Layer Decomposition

An important component of DDDDRRaW is Image Layer Decomposition (ILD) [13], the workload partitioning scheme. ILD is particularly well-suited for the challenges posed in Section 1 because under ILD: *(i)* the rendering problem at each node looks exactly the same as if it were an independent rendering application, facilitating the use of hardware rendering and all attendant optimizations (such as those involving pixel depth); *(ii)* using a small amount of preprocessing, we can predict the amount of data that each node must send per frame, allowing us to factor in the transmission time of each node when partitioning the work to minimize load imbalances; *(iii)* by replicating the scene on all nodes, we can avoid the need to communicate polygons, thereby decoupling the bandwidth requirement from scene complexity; *(iv)* each node needs only send the pixels that it has rendered, such that, in the best case, the bandwidth required is only proportional to the size of the final image and not the size of the NOW; and *(v)* there is no need to communicate Z-buffer information across the network.

We now give an overview of ILD because it has a strong effect on DDDDRRaW's implementation architecture and performance. Under ILD, the work of rendering scene objects is partitioned among all cluster nodes each frame. Given a set of scene objects to be rendered and a viewpoint, ILD assigns the objects in such a way that subsets of objects assigned to different machines are not mutually occlusive. For example, given the visibility ordering of the five objects A, B, C, D, and E shown in Figure 1(a), a legal ILD partitioning for a 3-node system is $\{A, B\}$, $\{D\}$, and $\{C, E\}$. Note that a strict ordering based on which subset is closer to the viewer is possible. In contrast, the decomposition $\{A, C\}$, $\{B\}$, and $\{D, E\}$ is not legal under ILD as A is closer to the viewer than B but B is closer to the viewer than $\{C\}$. Because objects in different subsets are not mutually occlusive, each node generates a coherent image layer; to compose the final image, ILD simply layers these image layers on top of one another according to the visibility order of the subsets (and so does not need the Z-buffer computed at each node).

More specifically, each frame, ILD partitions the overall rendering work for a system with P nodes as follows:

1. construct an occlusion graph as shown in Figure 1(a-b), where the vertices correspond to scene objects and a directed edge from A to B means that A may occlude B when viewed from the current viewpoint,
2. estimate the rendering time and *footprint* of each object, where the footprint of an object is defined as the set of pixels contained in its projection on the viewport, and
3. use the estimated object rendering times and footprints to partition the occlusion graph into P subgraphs such that: *(a)* if we collapse the occlusion graph so that there's a single vertex per subgraph as shown in Figure 1(b-c), the resulting *partition-occlusion* graph is acyclic, *(b)* the workloads defined by the subgraphs are balanced, and *(c)* the sum of the footprints defined by the subgraphs (which corresponds to the number of painted pixels that must be communicated) is minimal.

ILD currently targets hardware polygonal graphics accelerators. To make it possible to construct an occlusion graph and partition it in real-time, we use a static octree decomposition of the scene's world space to group the polygons into a relatively small number of aggregate objects (the leaves of the octree) [1]. This partitioning is controlled by two parameters: (1) each leaf octant should contain no more than $WorkThreshold$ percent of the total number of polygons in the scene,

[1] Because this decomposition is done statically, DDDDRRaW currently does not accept scenes with moving objects. We believe that it is possible to extend DDDDRRaW to handle scenes with limited numbers of moving objects [18]. Alternatively, there are other approaches for constructing the per-frame occlusion graph required by ILD (e.g., [8]).

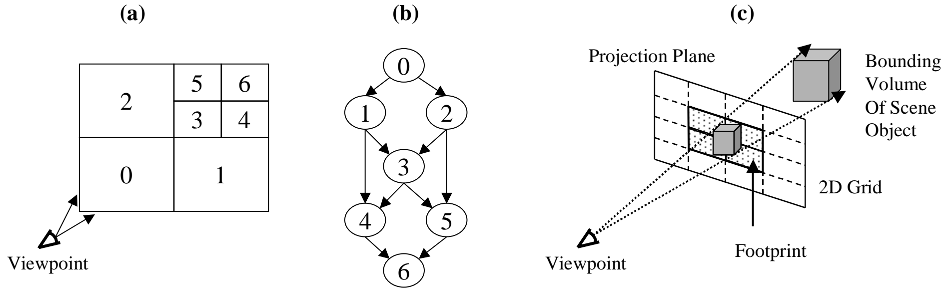

Figure 2: *(a) A quadtree decomposition of a 2D area and a particular viewpoint, (b) the corresponding occlusion graph, and (c) the computation of the footprint of an octant.*

and (2) each dimension of the bounding box of the polygons in a leaf octant should be no more than $SpaceThreshold$ percent of the corresponding dimension of the world space. A polygon is split into multiple sub-polygons if it intersects more than one leaf octant. Figure 2(a) shows an example partition albeit of a 2D area for ease of presentation. Given the leaf octants of an octree spatial decomposition, we use a simple visibility property of rectangular volumes to construct the occlusion graph [5, 13] (Figure 2(b)).

We estimate the rendering time of each aggregate scene object in frame f by measuring the object's rendering time in frame $f - 1$. We estimate the footprint of each object by projecting the bounding box of the polygons in it onto a course 2D grid representing the viewport as shown in Figure 2(c). The footprint of a set of objects is the union of the footprints of all member objects. The approximate transmission time of a footprint is the number of grid cells in the footprint times the cost of transmitting the pixels in a single cell.

We use the 2-phase greedy partitioning algorithm described in [13] for work assignment; this algorithm is not optimal but works well in practice. The scene description is replicated on all nodes before rendering begins so that assigning work only involves the transmission of (a small number of) object IDs.

4 Implementation

Any interactive 3D application built on top of DDDDRRaW consists of one *display* node and one or more *rendering* nodes as shown in Figure 3(a). On the display node, the application is linked with an instance of the DDDDRRaW library. At startup, the application is responsible for creating a window with the appropriate configuration for DDDDRRaW to render into and initializing DDDDRRaW with a VRML scene in the proper internal format. During normal execution, the application initiates the rendering of a frame by calling a DDDDRRaW entry point with a viewpoint and rendering state (e.g., lighting mode). DDDDRRaW renders the corresponding image and displays it in the given window.

In the case of our prototype VRML viewer, the viewer parses the VRML scene from a given file and computes the static octree decomposition at startup[2]. During normal execution, the viewer controls the application GUI, initiating the rendering of new frames in response to specific user events such as a mouse drag, which indicates the user's movement through the virtual world.

Each frame, on the display node, DDDDRRaW's specific actions include computing an ILD partitioning of the rendering work, communicating the new viewpoint and work assignment to each rendering node, receiving the image layers, and compositing and displaying the resulting image. On each rendering node, DDDDRRaW accepts the work assignment and renders and sends back the corresponding image layer. The communication of the image layer rendered for a frame may be overlapped with the rendering of the next frame.

Interestingly, computing an ILD partition involves the computation of each object's footprint, which is not a trivial computation. On the other hand, this computation allows us to quickly cull objects that are entirely out of the viewing frustum, saving the significant cost of transforming and culling each polygon in the object. Thus, DDDDRRaW will likely be particularly effective for large scenes where only a portion of the scene is visible in any given frame.

[2]Note that in a production system, we would expect scenes to be pre-decomposed to minimize the startup time.

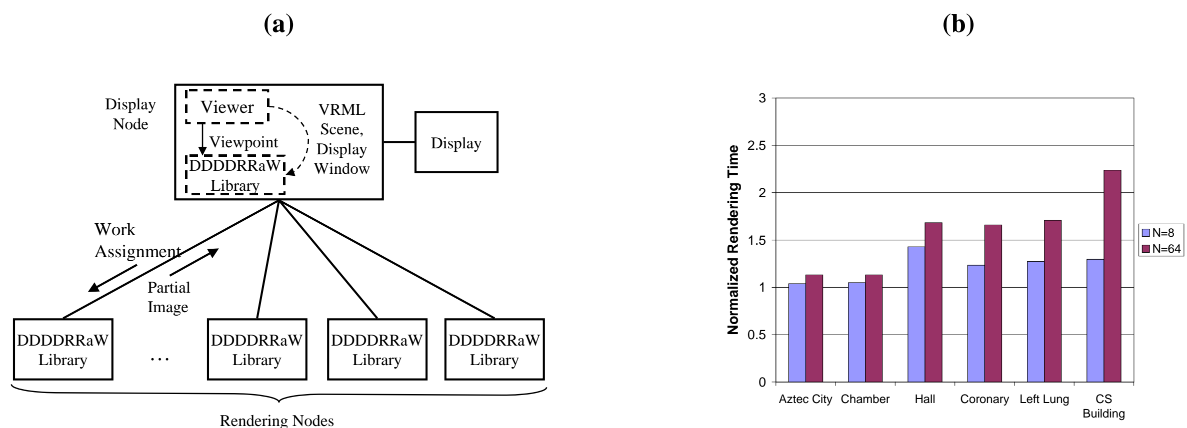

Figure 3: *(a) The generic architecture of any application that is built on top of DDDDRRaW. (b) Normalized rendering times of 6 VRML models on a 180 MHz R5000 SGI O2 for two octree decompositions (N is the number of leaf octants).*

4.1 Dynamic Selection of Octants as Scene Objects

In the original description of ILD, when decomposing a scene, we split a polygon if it is in more than one leaf octant. Polygon splitting can be expensive because it introduces extra vertices that must be transformed and lit. Figure 3(b) shows the normalized rendering times for six VRML scenes (see Table 1 in Section 5) when they are decomposed into 8 and 64 leaf octants. The rendering time for each scene is measured on a single 180 MHz Mips R5000 SGI O2 when viewed from the original viewpoint and is normalized against the rendering time of the un-decomposed version. For some scenes, this splitting does not result in very much overhead (implying that the polygons in these scenes are small and so not much splitting actually takes place) but for others, the overhead can be significant.

Splitting polygons is particularly expensive on the SGI O2 because the geometric transformation phase runs on the slow 180 MHz R5000 CPU, which is already the bottleneck for scenes with large polygon counts. We expect that the above performance degradation will be less serious on future machines that are better balanced—hardware vendors are already working on systems with specialized geometric transformation units and/or multiple processors to increase the speed of geometric transformation relative to rasterization[3].

Despite our expectation that the relative cost of splitting polygons will decrease, we are still motivated to reduce this performance degradation, especially since we may want to decompose to a fairly fine granularity for load balancing in the case where only a small part of the scene is visible to the user. In DDDDRRaW, instead of always using the leaf octants as the scene objects, we dynamically choose the appropriate level of decomposition from the octree at runtime. At startup, we statically decompose the scene into fine-grained objects using fairly small decomposition thresholds (e.g., $WorkThreshold = 10\%$, $SpaceThreshold = 15\%$). At run-time, we start by using octants from the first level of the octree that gives more octants than nodes (so that we can assign at least 1 octant to each node). Then, for each subsequent frame, if the ratio of the estimated rendering time of an octant rendered in the last frame to that of the entire frame is greater than a dynamic work decomposition threshold, we choose its children as the objects to be rendered for this frame. Likewise, if the ratio of size of the footprint of an octant to the total number of grid cells is greater than a dynamic space decomposition threshold, we choose its children. Of course, if the octant is an actual leaf of the original static decomposition, then we cannot dynamically choose its children. *We do not compute new sub-octants dynamically.* When dynamically decomposing, we estimate the rendering time of each child as $\frac{1}{8}th$ that of its parent. Figure 4(a-b) shows an example of this dynamic selection of octants as scene objects (again for a quadtree decomposition of a 2D area for ease of presentation).

On the other hand, if all the children of an octant were rendered in the last frame, we consider using this octant as the object to be rendered this frame instead of its children if: *(i)* the ratio of the sum of the expected rendering times of the children to that of the entire frame is below a dynamic work aggregation threshold, and *(ii)* the ratio of the size of the children's aggregate footprint to the number of grid cells is below a dynamic space aggregation threshold. There is a subtlety that should be considered when aggregating, however. When a number of the children of an octant are out of view while others are not, we may not want to aggregate back to the parent octant because this would reduce the effectiveness of culling. Thus, we do not aggregate back up to a parent octant unless enough of its children were in view last frame; in

[3]nVidia's GeForce 256 is a good example (http://www.nvidia.com).

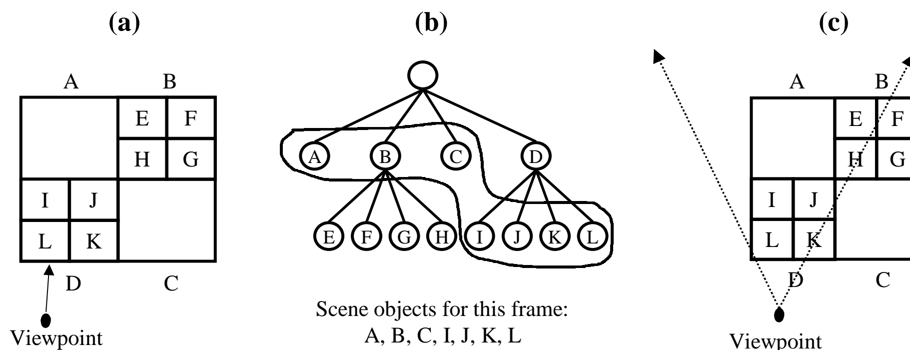

Figure 4: *Dynamic decomposition for a quad tree: (a) the static decomposition with 12 quadrants; (b) dynamically, our algorithm might choose to render I, J, K, and L since the viewpoint is close to these quadrants while choosing B because the viewpoint is far away; (c) G is completely out of view given the current viewing frustum. If we use E, F, G, and H as the scene objects in this frame, then G can be completely culled (when we are computing its footprint). On the other hand, if we use B instead, since B is partially in view, we have to transform all polygons in G to image coordinates before we know that they can be culled.*

our performance evaluation, we only aggregate to a parent octant if at least 5 of its children were in view. We determined this threshold empirically, measuring the average frame times of the test scenes/paths for a range of different values. When aggregating, we estimate the rendering time of an octant to be the sum of the expected rendering times of its children.

We use the following simple formulas to determine the dynamic decomposition and aggregation thresholds:

$$DecompThreshold = \frac{1}{MinNumOctantsPerRenderer * NumRenderers} \tag{1}$$

$$AggregateThreshold = \frac{1}{MaxNumOctantsPerRenderer * NumRenderers} \tag{2}$$

where $MinNumOctantsPerRenderer = 1.25$ and $MaxNumOctantsPerRenderer = 2.0$. $DecompThreshold$ and $AggregateThreshold$ are used for both work and space dynamic decomposition and aggregation. These parameters typically result in about 10 objects per rendering node, which should be sufficient for good load balancing [13].

4.2 Overlapping Communication and Computation

In DDDDRRaW's basic implementation, the display node blocks while waiting for the rendering nodes to render their assigned work. We can improve performance (as measured by average frame rate) by overlapping the communication of the image layers generated for a frame with the rendering of the image layers for the next frame. To effect this overlapping, we partition the functionality of DDDDRRaW on both the display and rendering nodes into a number of pipeline stages as shown in Figure 5; each pipeline stage is executed by a distinct thread. Although there are potentially seven stages in this pipeline, we currently use only three because *(a)* the render and compress stages both use the CPU heavily, and *(b)* the combined render/compress stage is the bottleneck and so further partitioning of stages cannot improve performance. (In fact, we really only use two stages, as we only allow two frames to be outstanding inside the pipeline. We view stage 1 and 3 as really being one stage because stage 2 dominates performance.) The viewer is allowed to initiate a new frame whenever one exits the pipeline.

4.3 Implementation Details

In addition to the dynamic selection of scenes objects described in the last section, several more implementation decisions affect DDDDRRaW's performance, including:

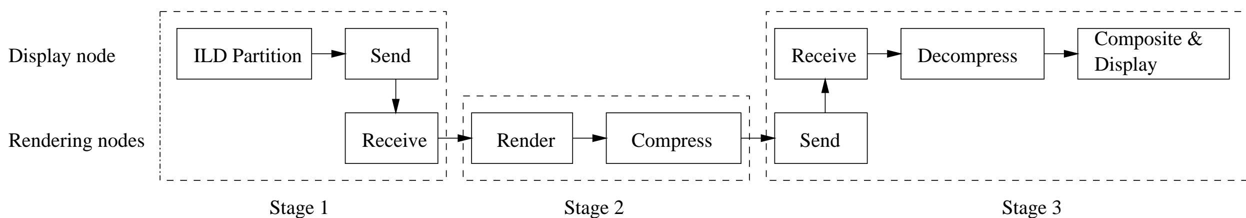

Figure 5: *DDDDRRaW's pipeline for overlapping communication with computation.*

- Because our current test-beds have relatively low network bandwidths, the rendering nodes compress each image layer before sending it to the display node. We use a software run-length encoding that has been used successfully in the past [3].
- The rendering nodes use UDP to communicate image layers to the display node. We implemented a thin layer of credit-based buffer management on top of UDP to reduce the possibility of packet loss—in the current prototype, message loss is a catastrophic event. Obviously, this would not be acceptable in a production system. We take this approach, however, since TCP can have unpredictable performance for the time granularity that we are interested in. We suspect that packets can sometime be lost (either in the switch or at the receiving node), causing slow-start to take effect.
- When computing the static octree decomposition of a scene, we often need to split polygons. For scenes that involve texturing or smooth shading (which accounts for all but one of our test scenes), polygon splitting involves computing new per-vertex texture coordinates and/or normal vectors. We have not implement this in DDDDRRaW because we are interested in system-level issues as opposed to rendering-related issues. Instead, we use OpenGL's dynamic clipping [23] to clip polygons that may protrude outside of an octant. This implementation shortcut comes at a cost: the overhead of dynamic clipping can be as high as 20-25% of the rendering time.

5 Performance

We begin evaluating DDDDRRaW's performance by measuring the execution time of significant components of DDDDRRaW to better assess where the performance bottleneck might be and DDDDRRaW's potential scalability. Then, since our ultimate goal is to improve the performance of real-time rendering, we compare DDDDRRaW's basic performance (in terms of average frame rate) against sequential rendering. Finally, we assess the impact of overlapping communication with computation on both frame latency and average frame rate.

5.1 Experimental Environment

We evaluate DDDDRRaW's performance on two clusters: a cluster of 5 180 MHz Mips R5000 O2s connected by a 100 Mb/s switched Ethernet LAN and a cluster of 8 750 MHz Athlon PCs connected by a 1Gb/s Giganet VIA LAN. The O2s have 100 MHz SDRAM while the PCs have 133 MHz SDRAM. On the O2s, geometric transformation is performed by the CPU while rasterization, z-buffering, and texture mapping run on a hardware accelerator. The PCs only support software rendering via the Mesa 3D Graphics Library [14].

The SGI O2's Unified Memory Architecture [17], where all computing subsystems have direct access to main memory via a 2.1 GB/s system bus, makes it an ideal machine to support distributed rendering. In particular, the frame buffer is kept in main memory, making it possible for a rendering node to send its rendered image layer to the display node with very little overhead beyond the communication cost itself[4]. Unfortunately, while the O2's memory architecture is ideal, its slow CPU and memory system limit the scalability of the display node and the achievable geometric transformation rate. This and the fact that our PC cluster is (somewhat) larger are the primary reasons why we also examine DDDDRRaW's performance on the PC cluster even though we do not have hardware accelerated rendering on this platform.

Our UDP-based communication layer achieves ~8 MB/s on the O2 cluster and ~40 MB/s on the PC cluster. Although the Giganet is a 1Gb/s LAN, the PCI implementation on our PCs seem to limit the raw VIA performance to ~45 MB/s.

[4]While we chose the O2s specifically because of this architecture, read and write bandwidth of AGP cards are increasing rapidly, making distributed real-time rendering on a PC platform plausible even though the frame buffer is not kept in main memory. Some PCs are even adopting architectures similar to that of the SGI O2, including the SGI visual workstations and PCs based on Intel's 810 chip-set and Dynamic Video Memory Technology [6].

Model	No. Prims.	Texture	No. Frames	Source
Aztec City	63933	N	100	ORC, Inc. http://www.ocnus.com
Chamber Hall	34537 22492	N N	62 58	Lightscape Technologies http://www.lightscape.com
Coronary Left Lung	19335 66876	N N	76 71	UW Structural Informatics Group http://sig.biostr.washington.edu
CS Building	56052	Y	138	Dept. of Computer Science, Millersville U. http://cs.millersv.edu/cs373.dir/vrml.dir

Table 1: *VRML models used for evaluating DDDDRRaW's performance. The reader can see what the models look like by visiting http://www.cs.washington.edu/XXX*

Display Node

- *ChooseDrawOctants:* Choose the octants to be rendered.
- *ConstructOcclusionGraph:* Construct the occlusion graph.
- *ComputeILDPartition:* Compute the ILD partition.
- *AssignWork:* Assign subsets of the partition to rendering nodes.
- *ReceiveImageLayers:* Receive image layers from rendering nodes.
- *DecodeImageLayers:* Decompress image layers and composite final image.
- *DisplayFrame:* Display final image.

Rendering Node

- *ReceiveAssignment:* Receive new work assignment from display node.
- *RenderImageLayer:* Render assigned work.
- *EncodeImageLayer:* Compress the generated image layer.
- *SendImageLayer:* Send the compressed image layer to display node.

Table 2: *Potentially significant per-frame operations at the display node and each rendering node.*

We evaluate DDDDRRaW's performance using fixed paths through six VRML models obtained from the web: Table 1 lists these models, and, for each model, gives the number of polygons, whether the model includes textures, the number of frames in the fixed path through that model, and its origin. All scenes were viewed in a 640x512 24-bit pixel viewport.

5.2 Micro-Benchmarks

Given DDDDRRaW's current architecture, Ahmdal's law states that the display node will eventually saturate and become the performance bottleneck. Thus, we first measure the major coordination components of DDDDRRaW to assess its scalability.

Table 2 lists the significant operations that must be performed each frame by the display node and each rendering node. Table 3 gives the measured execution time of these operations—except for those involving communication and rendering—for a predefined path through the Aztec City scene. Tables 4 and 5 give the network bandwidth, compression ratio, and average amount of data that must be sent. Note that the bandwidth is only approximate since it's hard to know exactly when data has been sent/received at the application level. We did not measure the cost of assigning work since it only involves sending several hundred bytes per node.

These measurements show that DDDDRRaW is effective at controlling the overlap in image layers as the size of the cluster increases, resulting in relatively slow growth in bandwidth and decoding time. They also show that on the O2s*: (1)* the cost of encoding and decoding are significant with respect to our target frame times of 33–100ms, limiting DDDDRRaW's scalability and achievable frame rate, and *(2)* the cost of computing object footprints dominates the cost of computing an ILD partition; since the cost to compute object footprints is proportional to the size of the system, it also limits DDDDRRaW's scalability.

Operation	Time (ms)													
	O2 Display Node			O2 Rendering Node			PC Display Node				PC Rendering Node			
	P=1	P=2	P=4	P=1	P=2	P=4	P=1	P=2	P=4	P=6	P=1	P=2	P=4	P=6
ChooseDrawOctants	0.09	0.09	0.62	–	–	–	0.04	0.04	0.08	0.12	–	–	–	–
ComputeFootprint	1.61	1.50	6.10	–	–	–	0.64	0.70	1.15	1.55	–	–	–	–
ConstructOcclusionGraph	0.25	0.23	0.95	–	–	–	0.09	0.11	0.16	0.24	–	–	–	–
ComputeILDPartition	0.13	0.15	1.01	–	–	–	0.07	0.08	0.14	0.19	–	–	–	–
ClearImageBuffer	3.50	–	–	–	–	–	4.36	–	–	–	–	–	–	–
DecodeImageLayers	18.08	22.84	30.28	–	–	–	2.66	2.92	3.24	3.24	–	–	–	–
DisplayFrame	0.18	–	–	–	–	–	7.88	–	–	–	–	–	–	–
CompressImageLayer	–	–	–	36.03	27.13	17.70	–	–	–	–	5.86	5.81	5.74	5.75

Table 3: *Measured execution time of non-rendering operations. These times were measured using a predefined path through the Aztec City scene. P is the number of rendering nodes.*

Model	1 Rendering Node			4 Rendering Nodes		
	Avg BW (MB/s)	Avg CR	Avg MB/f	Avg BW (MB/s)	Avg CR	Avg MB/f
Aztec City	7.17	29.05	0.04	6.87	35.67	0.07
Chamber	8.21	1.54	0.81	8.53	3.66	0.88
Hall	8.04	1.37	0.91	8.46	2.56	1.26
Coronary	8.07	5.02	0.23	9.06	11.00	0.29
Left Lung	7.99	2.76	0.30	8.84	5.90	0.45
CS Building	8.07	2.75	0.45	8.65	4.55	0.58

Table 4: *Average achieved communication bandwidth (BW), compression ratio (CR), and amount of data received at the display node for DDDDRRaW when running on 1 and 4 O2 rendering nodes.*

Model	1 Rendering Node			4 Rendering Nodes		
	Avg BW (MB/s)	Avg CR	Avg MB/f	Avg BW (MB/s)	Avg CR	Avg MB/f
Aztec City	38.27	0.35	0.05	49.20	0.83	0.08
Chamber	44.69	0.02	0.79	44.64	0.02	0.79
Hall	44.44	0.02	0.90	45.14	0.04	1.54
Coronary	42.06	0.11	0.13	52.82	0.36	0.17
Left Lung	43.38	0.50	0.30	51.04	0.16	0.39
CS Building	44.20	0.30	0.49	40.82	0.10	0.63

Table 5: *Average achieved communication bandwidth (BW), compression ratio (CR), and amount of data received at the display node for DDDDRRaW when running on 1 and 4 PC rendering nodes.*

Model	Average Frame Rate (fps)										
	O2 cluster				PC cluster						
	Sequential	Distributed			Sequential	Distributed					
		P=1	P=4	Max Speedup		P=1	P=4	P=6	P=7	Max Speedup
Aztec City	1.18	1.61	4.39	3.7	3.75	4.62	11.72	14.1	13.99	3.7
Chamber	0.97	0.94	2.39	2.5	3.43	3.37	8.18	9.84	10.22	3.0
Hall	1.34	0.71	1.60	1.2	2.55	1.95	4.80	5.70	5.98	2.3
Coronary	4.61	3.15	5.15	1.1	6.97	5.53	11.86	15.92	16.46	2.4
Left Lung	1.40	1.17	2.38	1.7	2.26	3.13	8.18	9.78	10.42	4.6
CS Building	0.66	1.62	3.45	5.2	0.70	1.03	2.60	3.66	3.98	5.7

Table 6: *Average frame rates for the VRweb viewer (sequential) and the basic DDDDRRaW implementation (distributed). P gives the number of rendering nodes.*

Observe, however, that the costs for encoding, decoding, and computing object footprints on the O2s are dominated by the slow CPU—these costs are significantly less on the PCs, implying that these operations scale well with increasing machine performance (not to mention the likelihood of having hardware acceleration for all of these operations in the near future). Furthermore, these costs are *not* proportional to the complexity of the scene. Thus, we can expect them to continue to track increasing performance of CPU, memory, and specialized hardware accelerators even as these more powerful machines are used to render ever more complex models.

In summary, ILD is effective (per its design goals) at controlling the overlap between image layers as the cluster size increases and DDDDRRaW can be expected to achieve modest scalability. Placing all coordination activities on the display node was an explicit decision that we made in trading off scalability for design simplicity and minimizing latency. One can easily imagine architectures where the rendering nodes cooperate to reduce the communication and compositing overheads on the display node (e.g., a tournament based communication/compositing structure). When there's little overlap between image layers, however, such tournament-based structures would increase the latency of completing each frame. Thus, instead of using such an architecture, we believe that the use of technologies that already exist today will allow DDDDRRaW to scale up to our targeted system size of 16–32 nodes as it is currently implemented.

5.3 Distributed vs. Sequential Rendering

We now turn to evaluating DDDDRRaW's actual performance by comparing its achieved frame rates to those of a (mostly) unmodified version of VRweb. While most of DDDDRRaW's rendering code derives from VRweb, there are several significant differences. DDDDRRaW has to pay the performance penalty of computing the ILD partition, polygon splitting, and dynamic clipping. On the other hand, DDDDRRaW can cull objects that are out of view when computing their footprints

Table 6 shows the average frame rates that VRweb and DDDDRRaW achieve for our test scenes. We give DDDDRRaW's average frame rates when running on only 1 rendering node to assess the impact of distribution overheads and object culling. We give DDDDRRaW's average frame rates when running on 4-7 rendering nodes to assess DDDDRRaW's ability to achieve better performance than the sequential renderer. Based on these results, we make the following observations.

DDDDRRaW-based interactive 3D applications can successfully exploit the resources of small commodity clusters to move closer to achieving interactive frame rates for complex scenes. On the O2s, despite the performance disadvantages that we have incurred to simplify the implementation and significant hardware limitations, DDDDRRaW achieves significantly greater frame rates for Aztec City and CS Building. Both of these scenes are large scenes, where only a portion of the scene is in view for most frames, so that object culling is very effective. Note that, in both cases, DDDDRRaW achieves higher frame rates than VRweb even when running on only 1 rendering node (but the speedup from $P = 1$ to $P = 4$ is still quite good, implying that culling is not the only significant source of performance improvement).

DDDDRRaW also achieves better frame rates for Chamber and Left Lung than the sequential renderer but less so then for Aztec City and CS Building. For Left Lung, this reduced efficiency is because polygon splitting imposes a significant performance penalty (see Figure 3(b)) and the entire scene is always in view so that object culling is ineffective. For Chamber, most of the scene is also in view for most of the frames so object culling is again ineffective. While polygon splitting does not impose a large penalty for Chamber, its communication requirement is the second largest of the scenes, and so imposes greater communication overhead.

DDDDRRaW only achieves a very slight increase in frame rates for Hall and Coronary. For Coronary, this is because the

Model	Avg MT Frame Rate	Avg MT Inter-Frame Display Time (ms)	Avg MT Frame Latency (ms)	Avg ST Frame Rate	Avg ST Frame Latency (ms)	Avg Seq Frame Rate
Aztec City	5.49	182	352	4.39	227	1.18
Chamber	3.58	279	542	2.39	419	0.97
Hall	2.39	418	826	1.60	625	1.34
Coronary	7.94	126	243	5.15	193	4.61
Left Lung	3.05	328	646	2.38	421	1.40
CS Building	5.05	198	378	3.45	290	0.66

Table 7: *Average frame rates, inter-frame display times, and latencies for the pipelined version of DDDDRRaW when running on 4 rendering nodes. We include previous results for the single-threaded version of DDDDRRaW and sequential rendering for comparison. MT stands for multi-threaded, ST stands for single-threaded, and Seq stands for sequential.*

single node rendering time of the scene is small (∼217 ms per frame). Compared to this rendering time, DDDDRRaW's overheads on the current test-bed is significant and so limits the achievable frame rate. In addition, polygon splitting is expensive for coronary and the entire scene is always in view so DDDDRRaW derives no benefits from object culling. For Hall, a combination of polygon splitting overhead, limited object culling, and poor compression ratio results in this limited performance improvement.

Performance trends are similar for the PC cluster. DDDDRRaW gets better speedup for Coronary and Left Lung because the overheads are lower on the faster PCs. DDDDRRaW gets better speedup for Hall because of the higher networking bandwidth of the VIA LAN. Interestingly, rendering performance is better on the PCs with software rendering than on the O2s with hardware rendering for all but the CS Building. This is because CS Building is the only scene with significant texturing, allowing the O2s to leverage their hardware accelerators more effectively. For other scenes, the geometric transformation is the bottleneck and so the faster PCs perform better.

DDDDRRaW seems to scale well with cluster size. Coupled with results from [13] and the discussion in Section 5.2, we expect that DDDDRRaW will scale to moderately-sized clusters, realizing our goal of providing rendering performance that is one to two generations ahead of what is possible on a single commodity machine. Except for the Aztec City, DDDDRRaW's speedup does not level out on our two clusters. While the overheads of encoding, decoding, and copying the final image to the frame buffer in X are all non-trivial, trends shown in Table 3 together with results from ***[13]*** provide a strong basis for projecting DDDDRRaW's scalability to clusters of at least ~16 nodes.

Object culling is a necessary performance optimization for large scenes (and so computing objects' footprints is not entirely a distribution overhead). The fact that DDDDRRaW outperforms the sequential renderer when employing only one rendering node for Aztec City and CS Building shows that, for large scenes, object culling is a necessary optimization. It is too expensive to cull individual polygons in such scenes (and will become more expensive as scene complexity increases). Thus, the necessity of computing objects' footprints in ILD, which only involves a bit more computation than what would be necessary for object culling, does not represent a significant distribution overhead.

5.4 Overlapping Communication and Computation

Unfortunately, Mesa 3D did not provide the necessary multi-threading support for our pipelined version of DDDDRRaW. Thus, in this Section, we only present and discuss results from the O2 cluster.

Table 7 gives the average frame rate, corresponding inter-frame display time, and frame latency for the pipelined version of DDDDRRaW. Inter-frame display time is the time period between the display of two consecutive frames. Frame latency is the time from when the VRML viewer initiates a frame in response to a user action until when it is displayed.

We make several observations based these results.

We can improve the average frame rate significantly by overlapping communication with computation. Overlapping communication and computation increases DDDDRRaW's average frame rates by 14–58%. This is as we expected since the communication and decompression costs are significant. By overlapping these operations with the rendering of a subsequent frame, we can increase the average frame rate. This is particularly noticeable for Coronary, where the sequential rendering time of the scene is not much greater than the distribution and communication overheads. DDDDRRaW now outperforms the sequential renderer by a factor of 1.8 whereas the basic implementation barely outperforms the sequential renderer at all.

The increase in average frame rate only comes with the trade off of significantly greater frame latency. While DDDDR-RaW's average frame rates increased by 14–58%, latency also increased by 18–59%. Two factors contribute to this increase in frame latency:

1. The communication of the image layers generated for the last frame takes place during the rendering of this frame. This increases the latency with which each frame is completed with respect to its initiation time since both rendering and communication use the CPU heavily.
2. With a pipeline depth of 2, the rendering of frame $f + 1$ is initiated when frame $f - 1$ has been displayed for some frame f. If the rendering time of frame f on each rendering node is greater than the total communication time for frame $f - 1$, then the portion that is not overlapped with the communication of $f - 1$ contributes to the latency of $f + 1$. We can avoid this effect if we do not initiate frame $f + 1$ until frame $f - 1$ has been displayed *and* the rendering of frame f has been completed.

Thus, if either the rendering or communication hardware is capable enough to not load the CPU, then overlapping communication and computation can be an effective optimization for distributed real-time rendering. Otherwise, we would only want to trade off latency, which affect the user's perceived system responsiveness, to improve frame rate if the latency remains under an acceptable threshold. While this threshold is subjective, 100 ms is typically accepted as a reasonable upper bound (e.g., see [2, 21]). This implies that we should not overlap communication with computation until we can achieve at least 10–15 fps (15 because multi-threading can increase latency as much as 58%), which is already at the boundary of interactivity. Beyond this, overlapping communication with computation can help to increase the average frame rate, making motion through the virtual world seem smoother to the user, and so may be worth the corresponding increase in latency.

6 Conclusions

We have described the prototype distributed rendering toolkit, DDDDRRaW, that we have developed in support of interactive 3D applications running on commodity hardware and a prototype VRML viewer built on top of DDDDRRaW. DDDDRRaW is a user-level library that exports the VRML rendering API. DDDDRRaW employs the on-line ILD partitioning and assignment algorithm developed in [13]. DDDDRRaW currently uses a simple run-length compression algorithm to overcome the limited bandwidths available on our test-beds.

We have evaluated DDDDRRaW's performance on two clusters. Our results show that the compression/decompression and communication time is currently a barrier to DDDDRRaW's scalability. Performance trends in moving from the slower O2 cluster to the faster PC cluster, however, show that with increasing CPU and LAN performance, DDDDRRaW should scale well to medium sized clusters (e.g., 16 nodes). When running on 4 O2 rendering nodes, DDDDRRaW's most basic implementation improves the average frame rate by a factor of 1.1 to 5.2 over sequential rendering for a number of test VRML scenes. When running on 7 PC rendering nodes, DDDDRRaW improves the average frame rate by a factor of 2.3-5.7. This increase in performance corresponds directly to lower latency as perceived by the user.

We have also evaluated a configuration of DDDDRRaW where communication is overlapped with computation to increase system efficiency. Our measurements show that this optimization can substantially increase the average frame rate (14–58%) but only by substantially increasing the frame latency as well (18–59%). Unlike traditional (scientific) distributed/parallel applications, this increase in the latency of intermediate computation steps may not be acceptable to users. Instead, we conclude that, without hardware support so that communication does not interfere with rendering, overlapping communication with computation should not be employed until the base system can achieve an average frame rate of at least 10-15 fps. This means that we cannot overlap communication with computation to help achieve an interactive frame rate! We can only use this optimization to increase motion smoothness once the base system can already achieve an interactive frame rate.

References

[1] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. A Framework for the Real-Time Walkthrough of Massive Models. Technical Report UNC-TR-98-013, Computer Science Department, University of North Carolina at Chapel Hill, 1998.

[2] M. Bajura, H. Fuchs, and R. Ohbuchi. Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery Within the Patient. In *Proceedings of the 19th Annual Conference on Computer Graphics*, July 1992.

[3] T. W. Crockett. Design Considerations for Parallel Graphics Libraries. Technical Report 94-49 (NASA CR-194935), ICASE, June 1994.

[4] T. W. Crockett and T. Orloff. A MIMD Rendering Algorithm for Distributed Memory Architectures. In T. Crockett, C. Hansen, and S. Whitman, editors, *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 35–42. ACM, Nov. 1993. Color plates on page 108.

[5] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.

[6] Intel Corp. Intel 810 Chipset: Great Performance for Value PCs. http://developer.intel.com/design/chipsets/810/810white.htm.

[7] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3), Sept. 1996.

[8] J. Lengyel and J. Snyder. Rendering with Coherent Layers. In *Proceedings of the 24th Annual Conference on Computer Graphics & Interactive Techniques*, pages 233–242, Aug. 1997.

[9] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[10] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Speed Rendering Using Image Composition. In *Proceedings of SIGGRAPH '92*, pages 231–240, July 1992.

[11] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Proceedings of SIGGRAPH '97*, pages 293–302, Aug. 1997.

[12] C. Mueller. The Sort-First Rendering Architecture for High-Performance Graphics. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pages 75–82, Apr. 1995.

[13] T. D. Nguyen and J. Zahorjan. Image Layer Decomposition for Distributed Real-Time Rendering on Clusters. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium*, pages 421–428, May 2000.

[14] B. Paul. The Mesa 3D Graphics Library. http://www.mesa3d.org/.

[15] M. Pichler, G. Orasche, K. Andrews, E. Grossman, and M. McCahill. VRweb: A Multi-System VRML Viewer. In *Proceedings of The 1st Annual Symposium on the Virtual Reality Modeling Language*, Dec. 1995.

[16] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh. Load Balancing for Multi-Projector Rendering Systems. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Aug. 1999.

[17] Silicon Graphics Inc. Unified Memory Architecture. http://www.sgi.com/o2/uma.html.

[18] O. Sudarsky and C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. In *Proceedings of EUROGRAPHICS '96*, pages C249–C258, Aug. 1996.

[19] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden Surface Algorithms. *ACM Computing Survey*, 6(1):1–55, Mar. 1974.

[20] The VRML Consortium. *VRML, http://www.vrml.org*.

[21] B. Watson, V. Spaulding, N. Walker, and W. Ribarsky. Evaluation of the Effects of Frame Time Variation on VR Task Performance. Technical Report 96-17, Georgia Institute of Technology, Graphics, Visualization, and Usability Center, 1997.

[22] S. Whitman. A Task Adaptive Parallel Graphics Renderer. In *Proceedings of the Parallel Rendering Symposium*, Oct. 1993.

[23] M. Woo, J. Neider, T. Davis, and D. Shreiner. *The OpenGL Programming Guide, Third Edition*. Addison-Wesley, 1999.