Image Composition Schemes for Sort-Last Polygon Rendering on 2D Mesh Multicomputers

Tong-Yee Lee, C.S. Raghavendra, *Senior Member, IEEE*, and John B. Nicholas

Abstract—In a sort-last polygon rendering system, the efficiency of image composition is very important for achieving fast rendering. In this paper, the implementation of a sort-last rendering system on a general purpose multicomputer system is described. A two-phase sort-last-full image composition scheme is described first, and then many variants of it are presented for 2D mesh message-passing multicomputers, such as the Intel Delta and Paragon. All the proposed schemes are analyzed and experimentally evaluated on Caltech's Intel Delta machine for our sort-last parallel polygon renderer. Experimental results show that sort-last-sparse strategies are better suited than sort-last-full schemes for software implementation on a general purpose multicomputer system. Further, interleaved composition regions perform better than coherent regions. In a large multicomputer system, performance can be improved by carefully scheduling the tasks of rendering and communication. Using 512 processors to render our test scenes, the peak rendering rate achieved on a 262,144 triangle dataset is close to 4.6 million triangles per second which is comparable to the speed of current state-of-the-art graphics workstations.

Index Terms—Sort-last-full, sort-last-sparse, polygon rendering, image composition, message-passing multicomputer system.

1 Introduction

Polygon rendering (Z-buffer) provides faster rendering speed, acceptable photorealism, and is supported by most commercial graphics workstations. A standard polygon rendering (Z-buffer) pipeline is shown in Fig. 1. There are two major processes occurring in this pipeline: namely, geometry (**G**) processing and rasterization (**R**) processing. Geometry processing starts by reading polygon primitives from an input database and then a modeling transformation is performed on each primitive to place each primitive on the right position of 3D space. After this transformation, the vertices of the polygons are illuminated by different light sources, transformed from 3D world space to 2D screen space, and truncated by a clipping pyramid. Some polygons can be clipped out without the need for further processing because they lie completely outside the viewing frustum. In the rasterization process, the colors of each pixel are calculated from the set of shaded 2D polygon primitives. The polygons are first scan converted to pixel values and then a Z-buffer hidden surface elimination is performed to determine their visibility.

Graphics rendering is computationally very intensive. With the massive volume of data being created by the scientists everywhere, there is a need to provide a faster rendering mechanism to visualize data. However, algorithmic improvements in rendering alone cannot meet the growing needs of scientific visualization. Fortunately, graphics rendering presents rich parallelism which is highly suitable for parallelization. There are two main approaches to exploiting the parallelism present in graphics rendering, namely, object based parallelism and image screen parallelism. Subsets of graphics primitives to be rendered, or regions of screen, can be partitioned among many processors. The first form is termed as object parallelism and the second form is termed as image parallelism [2]. By using parallel machines it is possible to significantly reduce the rendering time.

Parallel rendering is very useful for other reasons as well [3]. Scientific simulation datasets are generated on large parallel computers and their sizes range from hundreds of megabytes to hundreds of gigabytes. By performing graphics rendering on the same parallel machine, massive data shipment, often across a network with limited bandwidth, is avoided between that parallel machine and a graphics workstations. As a result, researchers can generate and visualize larger datasets, for example, by performing simulation for more number of time steps. Therefore, scientific simulation can be analyzed and visualized in more details to explore new phenomena.

There exist many parallel techniques for polygon rendering. Whitman's book [4] surveys multiprocessor rendering methods. In this paper, we will only present the most significant and recent work using parallel computers in the following section.

- *T.-Y. Lee is with the Department of Information Management, Nantai College, Tainan County, Taiwan, Republic of China.*
- *C.S. Raghavendra is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164.*
- *E-mail: raghu@eecs.wsu.edu.*
- *J.B. Nicholas is with the Environmental Molecular Sciences Laboratory, Pacific Northwest Laboratory, Richland, WA 99352.*

An earlier version of this paper appeared in the Proceedings of the Second Parallel Rendering Symposium, 1995.
For information on obtaining reprints of this article, please send e-mail to: transvcg@computer.org, and reference IEEECS Log Number V96024.

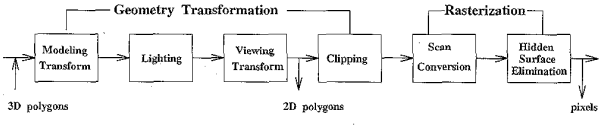


Fig. 1. Standard graphics pipeline.

2 Related Work on Parallel Polygon Rendering

There are several techniques to parallelize a polygon rendering algorithm. A straightforward approach is to directly map each stage of the pipeline into hardware [5]. Many commercial graphics workstations are built this way. Molnar et al. [6] describe a framework for parallel polygon rendering where the sort and redistribution of data occurs when transforming 3D objects (polygons) to 2D screen space (pixels). They delineate three types of parallel rendering algorithms: sort-first, sort-middle (image-oriented) and sort-last (pixel-oriented) (see Fig. 2).

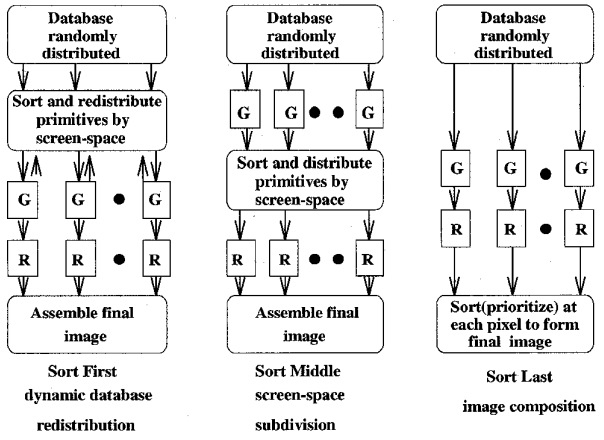


Fig. 2. Classification of parallel polygon rendering.

In the sort-first algorithms, each polygon is first preprocessed to determine which screen region it will be projected on. The primitive is then sent to the processor corresponding to this projected region, which performs all pipeline operations on that polygon. There is limited interest in the sort-first algorithms, because they are very vulnerable to load imbalances, caused by both geometry and rasterization processing. However, for adaptive image-space partition schemes, the information from a sort-last preprocessing pass can be used to alleviate some of the load imbalances [7], [8]. In addition, the coherence occurring between two consecutive frames can be exploited to reduce the polygon redistribution communication cost.

Most previous work focused on the sort-middle parallel algorithm. In this class of algorithms, each polygon's geometry transformation is first done locally, then the algorithm determines where the transformed polygon will be sent. The polygon redistribution is the main contributor to the communication cost. In the sort-middle algorithms, the geometry processing is generally balanced among the processors, while the rasterization part often causes load imbalances due to the nonuniform distribution of polygons on the screen.

Crockett et al. [9], proposed a sort-middle method on the Intel iPSC, which statically assigns consecutive scan-lines to processors. Many ways of overlapping the rendering execution time with the polygon redistribution time were investigated to reduce the overall communication cost. Also, they developed a performance model to determine the buffer size that minimizes the total communication overhead. No load balancing scheme was exploited in their scheme, and thus their parallel renderer incurs serious performance slowdown for nonuniform scenes.

Whitman [10] and Roble [11] investigated different adaptive screen division schemes to balance rasterization workloads. Both [10] and [11] complete the geometry transformations for all the polygons before starting to adaptively divide screen space (i.e., a global synchronization is required). Therefore, the number of polygons belonging to each region can be known before splitting the screen.

In Roble's work, the screen is recursively partitioned until each processor has an even rasterization load to perform. The number of polygons in a region is used as a heuristic to determine the amount of work in a region. This algorithm was implemented on an Intel iPSC hypercube. This method is good for smaller number of processors. However, when the number of processors increases and the assigned regions become small, the overhead in partitioning can be significant.

On the other hand, Whitman implemented his algorithm on a shared memory BBN TC2000 parallel computer. The load balancing is achieved by a task adaptive domain decomposition scheme which involves dynamic partitioning of rectangular pixel area tasks. The number of scanlines left to work is a heuristic for workload. When a processor finishes its assigned scanlines, it will steal half the number of scanlines from the maximally loaded processor. As larger processor configurations are used, the author's implementation analysis indicates that load imbalance is the major cause of performance degradation.

Ellsworth [12] proposed a method to balance rasterization workload for the current frame using information from the previous frame on the Intel Delta. Between two consecutive frames, a processor is responsible for gathering load information, computing load assignments (done via a greedy multiple-bin-packing scheme) and broadcasting assignments to other processors. Ellsworth also proposed a two-step sending scheme to reduce the number of messages when a large number of processors are used. The rendering rate is also extremely enhanced by writing the code in assembly language by which special graphics instructions as well as dual-instruction mode can be well exploited on the Intel i860 processor. As a result, this renderer achieves a very high rendering rate.

Another class of parallel rendering is the sort-last algorithm. Generally, this class of algorithms delay the data sort until the geometry processing and the rasterization of all polygons are completed. The polygons that constitute a scene are then evenly partitioned and each partition is assigned to a processor. After each processor finishes rendering its allocated polygons, the subimages created by the processors are merged into the final image. There are several possible variants of it. For example, the sort can be performed on-the-fly as pixels are generated, or can be performed incrementally as one portion of the screen is rendered before proceeding to another.

There were many previous efforts at the sort-last parallelism. A simple method is to order all N renderers to send

their subimages one by one to a destination composition processor. This technique was adopted in two commercial graphics architectures [13], [14]. Molnar [15] developed his high speed rendering system, called PixelFlow, using a pipeline style image composition scheme. In PixelFlow, there are N renderer processors. After all renderers finish their subimages, PixelFlow takes $N - 1$ pipeline steps to compose the subimages. The composed subimages at step r will be forwarded by renderer r to renderer $r + 1$ to compose with the subimage created in renderer $r + 1$.

An alternative scheme is to compose the subimages in a balanced tree [16], [17], [18]. In the composition tree, all renderers are located at leaf nodes. The composition procedure moves from the leaves to the root. Each composer at level i will compose all subimages from its children nodes at level $i + 1$. More processors become idle as the composition process proceeds to the root. Li and Miguet [19] proposed a different tree composition scheme on a transputer-based system, which can be reconfigured as a binary or ternary tree composition structure. Their polygon renderer was based on a scan-line Z-buffer scheme. Each processor performs both rendering and composition tasks. As rendering is completed, children nodes forward scan lines to their parent nodes, and parent nodes compose the incoming scan-line images. Both rendering and composition tasks are performed in a pipelined fashion along the tree. Finally, the root of the tree gathers the final composed image. The composition pipeline can be potentially slowed down by a few heavily loaded processors. The advantage of tree-merging schemes described above is that the completed image is available in its entirety at the root of the tree when the composing process completes. There is no need to gather subimages scattered among the processors.

Recently, Cox and Hanrahan [20] proposed a snooping protocol, adapted from the design of cache memory in tightly-coupled systems, to compose the subimages. Like Molnar's pipeline composition network, the snooping protocol requires $N - 1$ composition steps to combine all subimages from N renderers. However, in each step, one processor broadcasts only its "active" pixels in its local Z-buffer to all other processors. A pixel location is "active" at a given processor if at least one pixel has been rendered to it; otherwise it is "inactive." When other processors receive the broadcast, they will Z-buffer the local pixels, and the hidden pixels are deleted from the local active pixel list. Cox also developed a traffic model for this snooping scheme, and concluded that it can considerably reduce the composition data traffic.

As mentioned above, most sort-last techniques send the entire local Z-buffer data in each composition step [15], [18]. Molnar et al. [6] termed these techniques as sort-last-full algorithms. In contrast, algorithms are termed sort-last-sparse, if only active pixels are sent, as in Cox's snooping algorithm. They pointed out that the sort-last schemes need more sophisticated composition hardware or other schemes to reduce the large communication overhead, in particular, for supporting anti-aliasing by oversampling. Communication is difficult to scale well as system size increases.

Ma et al. [21] and Karia [22] described similar image composition techniques for parallel volume rendering. The so-called binary-swap composition technique was implemented on the fat-tree architecture of the CM5 [21]. Karia [21] implemented his scheme on the 2D mesh Fujitsu AP1000 parallel computer and termed it, the divide-and-conquer composition technique. Both techniques exploited bounding box optimization to speed up composition time and both are well suited for parallel polygon rendering (Z-buffer). Li et al.'s [25] perspective terrain renderer belongs to the sort-last-full class, and recently also used a binary-swap method in their implementation [26], but only merged the active pixels instead of a full image.

Ortega et al. [27] changed the standard graphics pipeline stages to fit the SPMD programming model on the CM-5. They used the virtual processor concept provided in the CM-5 to develop a data parallel polygon renderer. They achieved load balancing by continuously mapping the available data elements to the idle virtual processors. They also provided some rules to indicate if mapping is further required. Their polygon renderer belongs to the sort-last-sparse class. The individual pixels are routed to the right virtual processor which is in charge of that particular screen region. This routing is accomplished through the *sendmax* operator available on CM-5. Later, Hansen et al. also described this polygon renderer and the other two renderers, namely sphere renderer and volume renderer in [28]. On CM-5, they used the CMMD function *CMMD_reduce_v*, with a minimum operator, to compose all Z-buffers in logarithmic time for the sphere renderer. As for the volume renderer, they exploited binary-swap method for composing images.

In the following sections, we will first outline our parallel polygon renderer built on the 2D mesh Intel Delta parallel computer. Then we describe our sort-last-full image composition scheme and another well-known divide and conquer composition technique (binary-swap composition) proposed by Ma et al. [21] and Karia [22]. Our composition scheme extends recent advances in global communication algorithms proposed by Barnett et al. [29] for 2D mesh parallel architectures. We experimentally evaluate both schemes and show that our scheme is generally superior to binary-swap composition technique on a 2D mesh parallel computer, such as the Intel Delta. There are several alternatives based upon our sort-last-full scheme and these are presented and experimentally evaluated. For our experiments, we used five scene models from the public domain dataset SPD [30], with sizes ranging from 150K to 500K triangles. Using 512 processors of the Delta, we achieved a peak rendering rate of about 4.6 million triangles per second on a 262,144 triangle dataset. Finally, some concluding remarks and future works are given.

3 Overview of Our Sort-Last Rendering System

In our parallel renderer, each processor does both geometry transformation and rasterization. A global image composition process is exploited to merge all subimages generated by the processors. In this section, we first discuss how the polygon dataset is partitioned and distributed among the processors, then present some details of our polygon renderer.

3.1 Dataset Partition and Distribution

The first step in our parallel rendering task is to partition the polygons in the scene and distribute them among processors. Assuming that we have N processors and T triangles in a test scene, there are two common ways to assign T/N triangles to each processor: we can distribute triangles in an interleaved manner like [9] among the processors; or we can assign first T/N triangles to the first processor, the next T/N triangles on the second processor, etc. We call the first scheme interleaving assignment and the second group assignment. In most databases like SPD, groups of triangles stored near each other generally lie near to each other in the scene as well. Therefore, group assignment tends to keep triangles near each other in the scene to appear in the same processor. In such situations, we are likely to use tight bounding box to contain assigned triangles. For image composition, tight bounding box will not cause too much redundancy in both communication and computation for the sort-last-full system. However, the highly localized distribution of triangles in the screen space can cause load imbalances in rendering phase among the processors. Interleaving, on the other hand, is prone to make the distribution of triangles on the screen similar for each processor and thus potentially evens out imbalances during the rendering phase. But, interleaving tends to make triangles near each other in the scene to appear in different processors and thus potentially leads to redundancy in composition time for the sort-last-full system. To take advantages of both schemes, the group interleaving is used in our implementation to distribute triangles and is described below.

For the group interleaving, the whole database is partitioned into a few larger groups first. In our implementation, each larger group contains 2,000 triangles for our test scenes. Then, this larger group of polygons are evenly partitioned into N (0, 1, ... $N-1$) small groups and distributed among the processors as follows: Processor 0 reads a large group of polygons from the disk. Processor 0 keeps the 0th small group of polygons from it and forwards the rest to other processors. Processor P_i picks up the ith contiguous small group. After reading a small group to its memory, processor 0 asynchronously reads the next large group from the disk and repeats the process until all polygons have been imported from the disk. At the end of this data distribution process, all small groups of polygons contained in a dataset are evenly distributed among processors in an interleaved manner. We term this data distribution process as "group interleaving." In a later section, we will show its effect on the rendering performance.

3.2 Our Polygon Renderer

Many parallel renderers have been proposed in the past. Some were written in assembly language [12] and some pre-computed many parameters, such as normal vectors [9], [12] and shading [9], for each triangle before rendering. These factors can affect the rendering performance significantly. To compare the performance of various schemes, we need to clearly indicate what we compute in our renderer. In our parallel renderer each processor creates a full screen image on its local Z-buffer memory and renders its assigned polygons as follows:

Rendering Loop
for each frame image *do*
0.: compute view and transform matrices, and initialize Z-buffer.
for all local polygons *do*
1. : compute normal vector for current polygon.
2. : do backface culling on this polygon.
3. : do lighting on each vertex of polygon.
4. : do perspective transformation.
5. : do clipping.
6. : do scan conversion.
7. : Z-buffer this polygon.
enddo
8. : globally compose image.[1]
9. : clear Z-buffer.
enddo

In our implementation, we optimize for speed, therefore, we use simple Gouraud shading. In Gouraud shading, we use the colors of each vertex (computed in step 3) to linearly interpolate the colors on the polygon in step 6. After each processor finishes rendering the local polygons, we do a global composition of all subimages to obtain the final image. Our resulting image has 512×512 resolution without anti-aliasing. All codes are written in the C language.

4 Parallel Composition

In the last section, we reviewed many previously proposed sort-last polygon rendering algorithms. In most algorithms [15], [16], [17], [18], [19], more and more processors become idle as the composition proceeds—a waste of computational power. In contrast, Ma et al. [21], Karia [22], Wittenbrink [23], and Wittenbrink and Harrington [24] presented the divide-and-conquer image composition techniques, where processors are kept busy as much as possible, to improve parallel volume rendering. The divide-and-conquer method (binary-swap) used in [21] and [22], is well suited for parallel polygon rendering (Z-buffer). In this section, we describe the binary-swap technique, and propose another divide-and-conquer scheme.

4.1 Composition by Binary-Swap (BS, Sort-Last-Full)

The aim of the binary-swap (BS) composition is to exploit more parallelism in the composition stage and to keep every processor involved in all stages of the composition process. In the BS scheme, only half the image is swapped between a pair of processors and each processor pair composes the two opposite halves of subimages at each composition stage. As the composition proceeds, the processors are responsible for smaller and smaller portions of image composition. In total, the BS composition requires $\log N$ composition stages, and each processor keeps a fraction $(1/N)$ of the final image, where N is the number of processors in use.

Fig. 3 shows an example of BS using four processors. Each processor's Z-buffer is divided into four disjoint areas Z_{00}, Z_{01}, Z_{10}, Z_{11}, where $Z_{**} = Z_{00} \cup Z_{01} \cup Z_{10} \cup Z_{11}$. In the

1. In some of our implementations, part of step 8 can be done inside the second *for* loop.

first stage, processor, 0 (P_{00}) sends its Z_{1*} (i.e., $Z_{10} \cup Z_{11}$) to its neighbor processor 1 (P_{01}) and receives a Z'_{0*} from processor 1. Conversely, processor 1 sends its Z_{0*} to processor 0 and receives a Z'_{1*} from processor 0. Both processors complete this "send and receive" and then compose the received subimage with the local subimage. Meanwhile, processors 2 (P_{10}) and 3 (P_{11}) do a similar exchange of subimages. In the second stage, processor 0 (P_{00}) sends its local Z_{01} to processor 2 (P_{10}) and receives Z_{00} from processor 2, where this incoming Z_{00} was composed earlier by processors 2 and 3 in stage 1. Processor 2 does the converse operations in the opposite direction. Similar subimage exchanges and compositions occur between processors 1 and 3. At the end of the composition phase, each processor holds a final composed subimage equivalent to $(1/N)$ of the final image.

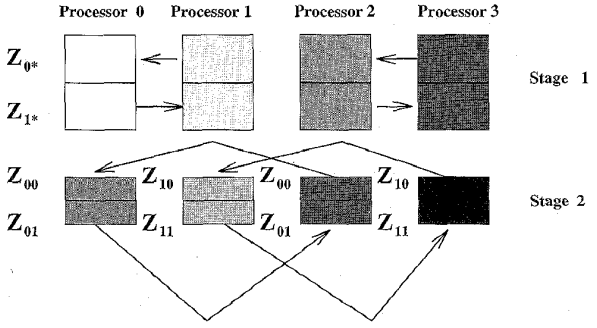


Fig. 3. Binary-swap using four processors.

4.2 Version 0: Composition by Parallel Pipeline (PP, Sort-Last-Full)

In our parallel pipeline (PP) approach, the Z-buffer at each processor is divided into N (Z_0, Z_1 ..., Z_{N_2}, Z_{N_1}) portions for subimages. Processors are organized on a circular ring and are denoted as P_0, P_1 ..., P_{N_2}, P_{N_1}. P_{next} and P_{prev} for a processor P_i will be $P_{(i+1) \bmod N}$ and $P_{(i\_1) \bmod N}$. To circulate and compose the subimages along the ring using the following algorithm, $N-1$ stages are necessary.

Parallel Pipeline Composition

for all processors *do* in parallel
1. : set current composed area $B_{current}$ as Z_i in processor P_i
2. : *for* j = 1 *to* $N-1$ *do*
3. : Each processor sends $B_{current}$ to its P_{next} processor
4. : Each processor receives a $B'_{current}$ from its P_{prev} processor
5. : set $k = (i-j) \bmod N$
6. : Each processor composes its incoming $B'_{current}$ with its local Z_k
7. : set newly composed Z_k as $B_{current}$

enddo
enddo

The subimages are accumulated in a pipelined fashion along the ring, with each processor involved in each stage. At the end, each processor P_i holds a fraction of the final image at partition $Z_{(i+1) modN}$. Fig. 4 shows an example of our scheme using three processors. The Z-buffer in each processor is divided into three disjoint parts, Z_0, Z_1, and Z_2. In stage 1, processor P_0 sends its Z_0 to processor 1 and receives a Z'_2 from processor P_2. Processor P_0 composes Z'_2 with its local Z_2 to form a new Z_2. In stage 2, processor P_0 sends its new Z_2 to processor P_1 and receives a Z'_1 from processor P_2 to compose with its local Z_1. The resulting new Z_1 in processor P_0 is a portion of the final image. Similar subimage exchanges and compositions occur in a pipelined fashion between processors P_1 and P_2.

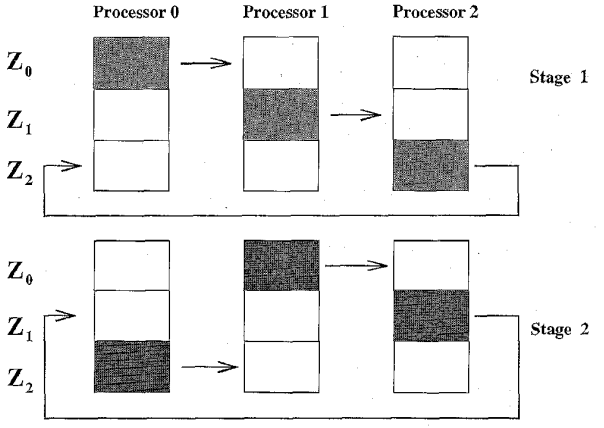


Fig. 4. Parallel pipeline using three processors (1 × 3).

For 2D mesh parallel computers, like the Delta and Paragon, we logically group the 2D mesh ($r \times c$) into many sub-rings. In the first phase of our algorithm, the PP composition is executed along one dimension, say within each column independently (as a sub-ring of r processors). On each processor, the local Z-buffer is divided into r equal subimages. It takes $r-1$ steps to circulate the subimages along the ring and to accumulate the result in a pipelined fashion to produce temporary subimages distributed among the processors. After the first phase, each processor holds a temporary subimage that contains the accumulated result along the entire column. In the second phase, a similar composition process is repeated, but now along each row independently (as a subring of c processors) using the subimage that all of the processors in that row share in common as the entire image. At the end of the second phase, the image has been composed, with the final image being distributed among all N processors. In total, our algorithm takes $r + c - 2$ steps to form a final image on the 2D mesh parallel computers.

4.3 Analysis and Experimental Evaluation

Before we evaluate both the BS and PP parallel image composition techniques, we need to point out that similar techniques have been used for global vector combining [29]. Barnett et al.

[29] named their techniques recursive-halving (similar to the BS) and a bucket scheme (similar to the PP). Here, we experimentally evaluate both schemes on the 2D mesh, assuming that a full image data of Z_i is sent in each stage.

In both schemes, the amount of data transferred per processor is approximately the same (about $\frac{(N-1)\times Z}{N}$ and Z is the size of image screen). This differentiates between the BS and PP methods from the balanced tree algorithms. The tree algorithms require processors to send an amount of data equal to a full Z or many times Z [15], [18]. We note that the BS takes optimal (i.e., log N) stages in composition and the PP takes more stages (i.e., $r + c - 2$). However, in the BS, there will be many number of messages contending for a single network communication link on a mesh-connected architecture. Unlike in hypercubes, the BS cannot avoid contention in 2D mesh because there are not enough communication links. With image resolution ranging from 64×64 (low resolution) to $1{,}024 \times 1{,}024$ (high resolution), a full image might take 32K to 8M bytes. When this amount of or larger messages transfer on the links or more processors are used, the saturation on the links will degrade BS's communication performance severely. In contrast, larger startup costs (i.e., $r + c - 2$ steps) in PP becomes less significant within or beyond this range of image resolutions.

Next, we experimentally evaluate the above observation and show that the PP scheme is better than the BS scheme, using different resolutions and mesh sizes, even though the PP needs more composition stages. In our implementation, each pixel value is eight bytes long and consists of a float z depth value (4 bytes) and a color quadruple (red, green, blue, *alpha*; 4 bytes). In practice, we found that the bandwidth of communication network in y direction is greater than that in x direction on the Delta. Therefore, we prefer performing composition in the y direction first for larger data (i.e., $\frac{Z}{r}$) and then the x direction (i.e., $\frac{Z}{N}$). We performed our experiments for different mesh sizes and image resolutions ranging from 64×64 to 512×512. Fig. 5 shows the composition timings for different image resolutions using mesh sizes of 16×32 and 4×4. This figure clearly indicates that the PP performs better than the BS. For the PP, performance ranges from 0.02 to 0.55 seconds per image. In comparison, the BS scheme requires 0.03 to 1.7 seconds for the same size of images. The BS scheme performs well only at lower resolution (i.e., less than 64×64) images.

On the Delta, we can simply model the communication cost to send and receive a message of L bytes between two processors at any distance by $\alpha + \beta_d L$, where α is the startup latency per message, β_d is the transfer time per byte in the d direction (i.e., x or y). The cost of PP is defined as follows:

$$Cost_{pp}(Z,N) = \sum_{i=1}^{r-1}\left[\alpha + \frac{8Z}{r}\beta_y + \frac{Z}{r}\gamma\right] + \sum_{i=1}^{c-1}\left[\alpha + \frac{8Z}{N}\beta_x + \frac{Z}{N}\gamma\right] \quad (1)$$

Assuming $\beta_x \approx \beta_y$, the typical values reported in [31] are $\alpha = 157\mu$ seconds and $\beta_x = 0.21\mu$ seconds. As described above, each pixel contains 8 byte value, for which we measured $\gamma = 0.58\mu$ seconds per pixel, where it includes a depth comparison and a 4 byte color quadruple assignment. Table 1 shows composition timings for rendering a 512×512 image size using different numbers of processors. This table shows that the cost of PP is independent of the number of processors and the predicted values from (1) accurately match our experimental results. For example, under 512 processors, our experimental timing is about 0.59 seconds and the prediction of (1)is 0.58 seconds. Therefore, (1) can provide a good prediction for the PP scheme and an upper bound for other sort-last-sparse implementation based on PP.

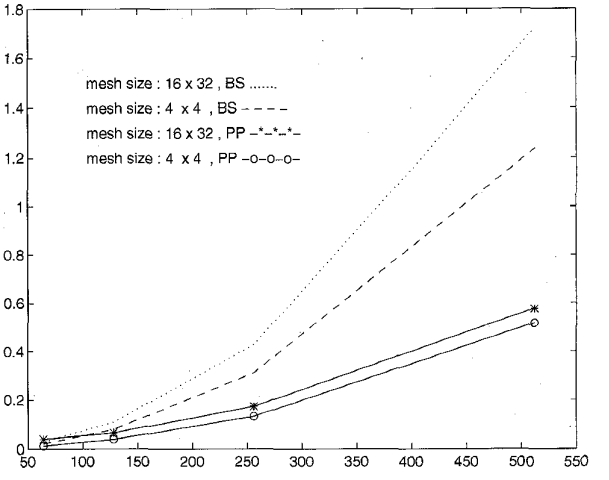


Fig. 5. Composition timings for rendering different image sizes using 16×32 and 4×4 mesh sizes.

TABLE 1
THE IMAGE COMPOSITION TIMINGS (SECONDS) AT 512 ´ 512 RESOLUTION FOR BS, PP, AND THE PREDICTED VALUES BY (1)

#procs	16	32	64	128	256	512
BS	1.23	1.43	1.48	1.67	1.69	1.71
PP	0.51	0.54	0.55	0.56	0.57	0.59
Predicted	0.53	0.56	0.56	0.57	0.57	0.58

4.4 Version 1: Optimization Using a Bounding Box (PPB, Sort-Last-Full)

In this subsection, we will present a scheme termed PPB in which several bounding boxes are used to optimize image composition. In the BS scheme, each processor is responsible for large image areas in the early stages; however, pixels are sparse in these areas. This sparsity in composition area decreases as the composition proceeds, since more processors contribute to each area. On the other hand, in the PP method, since each composed area is bounded by $O(Z/r)$ in the first $r - 1$ stages and by $O(Z/N)$ in the last $c - 1$ stages, the sparsity at each stage is relatively less than that for the BS scheme.

We can avoid sending "inactive" pixels if we can look up an arbitrary active pixel very quickly, and determine the amount of active pixels on the fly. Special memory access hardware is usually necessary for this purpose [20]. Ma et al. [21] suggested using a bounding box at each composition stage to include all active pixel areas. Each processor binary-swaps pixels only within this box. This technique works very well for volume rendering, since local subimages are rendered from a block of continuous voxel data. In the PP scheme, the local Z-buffer is divided into many fixed portions (i.e., Z/r or Z/N). In our implementation, we used a single bounding box for each portion at each composition stage. For example, in Fig. 6, our implementation used two smaller bounding boxes for these two disjoint parts (Z_0 and Z_3) and empty bounding boxes for Z_1 and Z_2. We call this implementation PPB scheme.

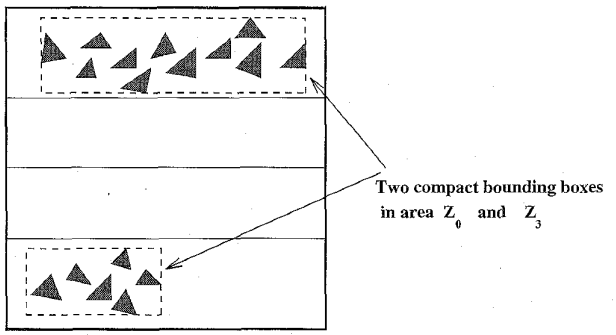


Fig. 6. More compact bounding boxes for different areas in the parallel pipeline method.

In general, a bounding box is not a single contiguous block of memory; rather, it is composed of several smaller contiguous strips of pixels, one strip for each scanline which intersects a bounding box. To communicate the contents of the bounding box to another processor, additional overhead in data copying or setup is required. These extra costs include copying the contents of the bounding box into a contiguous buffer and handling of multiple composition buffers. Later, we will experimentally show that these costs do not offset the performance with bounding box optimization, but yield better rendering rate than PP does.

4.5 Version 2: Direct Pixel Forwarding (DPF, Sort-Last-Sparse)

For the PP with the bounding box, the subimages are accumulated along each dimension of the 2D mesh. The size of the bounding box can grow gradually, while its upper bound is O(Z/r) or O(Z/N). Here, we present a direct pixel forwarding (DPF) scheme without sending sparse pixels. In the DPF, a processor at each stage directly sends a subimage Z_i to that processor where the final Z_i is stored. The sending sequence is ordered to avoid link contention. The DPF composes the subimages using the following algorithm.

Direct Pixel Forwarding Composition

for all processor P_is *do* in parallel
1. : *for* j = 1 *to* $N-1$ *do*
2. : Each P_i sends its $Z_{(i+j+1) \bmod N}$ to processor $P_{(i+j) \bmod N}$
3. : Each P_i receives a $Z'_{(i+1) \bmod N}$ from processor $P_{(i-j) \bmod N}$
4. : Compose the local $Z_{(i+1) \bmod N}$ with the incoming $Z'_{(i+1) \bmod N}$
enddo
enddo

In the DPF scheme, each processor keeps a fraction of the Z-buffer and maintains many active pixel queues for pixel forwarding. When an "active" pixel does not belong to the local Z-buffer, this "active" pixel information including x, y coordinates, color quadruple, and z depth value will be inserted into the corresponding pixel queue. In each composition stage, the processor sends one active pixel queue out to the corresponding processor for composition. Fig. 7 shows an example of the DPF using 1×4 processors. In stage 1, processor P_0 sends its active pixel information located in Z_2 to processor P_1 and receives one pixel information containing Z'_1 from processor P_3. Processor P_0 composes Z'_1 and its local Z_1. In stage 2, P_0 sends its Z_3 to processor P_2 and receives a Z'_1 from P_2. In stage 3, P_0 sends its Z_0 to processor P_3 and receives a Z'_1 from P_1. Similarly, the active pixel information exchanges and compositions occur in this fashion for processors P_1, P_2, and P_3.

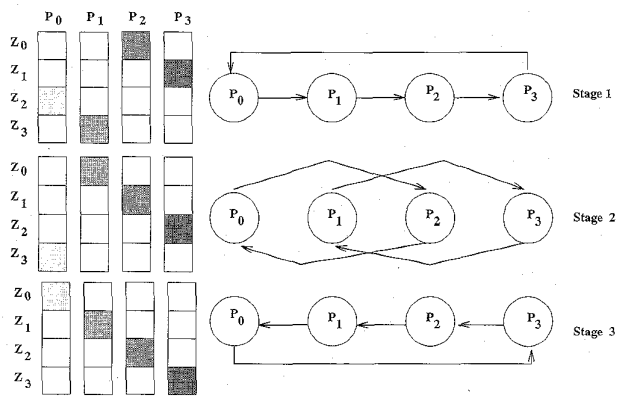


Fig. 7. Direct pixel forwarding composition using four processors (1×4).

Again, for 2D mesh parallel architectures, we logically group the 2D mesh ($r \times c$) into many sub-rings. In the first phase, the DPF is executed along the y dimension which takes $r-1$ steps. It takes another $c-1$ steps along the x dimension in the second phase. A variant of the DPF is that we can limit the length of any active pixel queue, and if any active queue is full, during rendering, we can send out the active pixels asynchronously to reduce the amount of active pixels to be transferred before the first phase. This variant will be described in detail in a later section. In fact, according to Cox and Hanrahan's [20] and Tay's results [32], the probability that local pixel depth exceeds 1 is small. Therefore, we can safely send pixels out in the first phase because the local Z-buffering will not reduce data traffic very much. On the other hand, as processors receive active pixel information from other processors during rendering time or in the first phase, we need to Z-buffer them and cannot send data out until the second phase starts. This intermediate Z-buffering can reduce some data traffic occurring in the second phase. An example of the intermediate Z-buffering is shown in Fig. 8. In this example, processors P_0, P_2, and P_3 send A, B, and C numbers of pixels, respectively, to P_1 after the first phase. The number of final composed pixels (A, B, C, and D) is at most equal to their summation, and is always expected to be less due to the overlapping of some Z values. The more processors that contribute to the composed area, the higher the probability that pixel depth exceeds one. Therefore, in the second phase, the amount of data transferred can be reduced. The amount of reduction is dependent on the average original pixel depth.

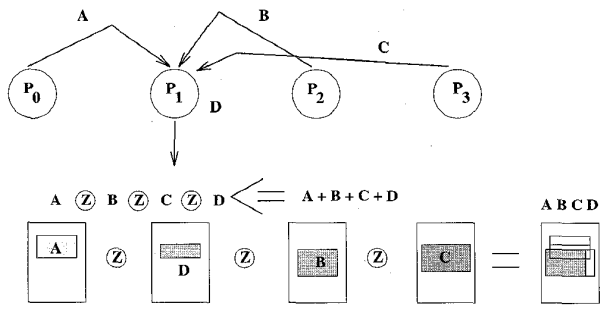


Fig. 8. The effect of intermediate Z-buffering on data traffic.

There is a sort of unifying principle that's shared between the DPF and Ellsworth's two-step sending idea [12]. In both cases, the buffered data goes to an intermediate node before being forwarded to its final destination, and in both cases the communication complexity is reduced by not sending the data directly to its final destination. However, there are several major differences between these two techniques. First, we are sending buffers of pixels instead of sending buffers of triangles as Ellsworth did in his sort-middle renderer. Second, our composition steps are along rows and then along columns, whereas his were between rectangular regions of processors and then within regions. Third, in his two-step sending method, the polygons are sent twice, instead of once, before they reach the final destinations. This means the number of polygons transmitted is doubled. However, as described above, the intermediate Z-buffering in our DPF scheme can reduce some data traffic occurring in the second phase. So, the number of pixels transmitted can be less than a factor of two.

4.6 Version 3: DPF with Static Load Balancing (DPFL, Sort-Last-Sparse)

In the DPF scheme, data exchanges and pixel compositions can be unbalanced due to uneven active pixel distribution. We can alleviate this problem to some extent by using a static load balancing and we call this the DPFL scheme. In the first phase, we can assign horizontal lines among the processors in an interleaved fashion. This is followed by assigning vertical lines among the processors in the second phase. In a 16×32 2D mesh Delta and for a 512×512 or higher resolution images, the number of processors in each dimension is not high, and thus processors have enough interleaved lines to even out the imbalances in each phase. From our experience, we only require interleaving the scan-lines in the first phase. Two phase interleaving does not make much difference in the speed of composition. Use of interleaving to provide load balancing in the sort-last-sparse strategies like the DPFL scheme has been mentioned in the literature [6]. In a later section, experimental results will show that the interleaved composition regions perform better than coherent regions (consecutive scanlines) in our implementation.

4.7 Version 4: DPF with Task Scheduling (Sort-Last-Sparse)

In this subsection, we present a sort-last-sparse rendering scheme termed DPFS (DPF with task scheduling between communication and rendering work) with an attempt to reduce the communication time. The task scheduling scheme presented here is similar to that in [9]. We separated the rendering computation completely from the global pixel composition in earlier schemes. Pixel merging cannot begin until each processor has rendered all local polygons. For large system, such a disjoint approach leads to large messages must be sent at about the same time. This will likely leads to high communication overhead [9]. To reduce message sizes, we can schedule both communication and rendering work by overlapping them as was done in [9]. In addition, all earlier schemes under utilize the communication links on the Delta. Each phase of earlier versions exploits communication links only in one direction (x or y dimension separately). Based on the above observations, the scheme termed DPFS is proposed and a pseudocode version of it is described as follows:

DPFS

```
while (local triangles are not yet rendered)
{
select a local triangle;
render it into the A-type or the B-type buffers if its
rendered pixels are outside local processor's
assigned region, and send buffer if full;
if incoming messages exist
for each incoming message
{
if message needed to be forwarded in the second phase of
the DPF
then unpack this message into the B-type buffers;
else Z-buffer this message with local region;
}
}
flush all A-type buffers to other processors in the order of
DPF's first phase;
while (the A-type messages remain to arrive from other
processors)
{
if message needed to be forwarded in the second phase of
the DPF
then unpack this message into the B-type buffers;
else Z-buffer this message with local region;
}
flush all B-type buffers to other processors in the order of
DPF's second phase;
while (the B-type messages remain to arrive from other
processors)
{
Z-buffer this message with local region;
}
synchronize; /* make sure all processors finish this frame */
```

In the DPFS scheme, there are two types of message buffers which consist of $(r - 1)$ A-type message buffers and $(c - 1)$ B-type message buffers. The A-type message buffers

store both pixel values and (x, y) coordinates of corresponding regions in the first phase of the DPF scheme, and the B-type message buffers are in charge of the second phase. In the first *while loop*, the rendered pixels of each local triangle can be temporarily stored either in the A-type buffers (i.e., these pixels do not belong to local processor's region in the first phase of the DPF) or in the B-type buffers (i.e., these pixels do not belong to local processor's region in the second phase of the DPF) or can be Z-buffered in local processor's assigned portion of the final image. We implement our scheme by asynchronous routines for message send and receive, and these can be used to overlap message transfers with both triangle rendering and pixel merging computations. We can find these overlappings within these three while loops. For example, in the second while loop, if there are still A-type messages remaining we post an asynchronous receive at once after a message is received, and then deal with this incoming message. Therefore, by this means, we hope to overlap communication time with computation as much as possible. These overlappings do not exist in the earlier schemes. Since both the A-type (traveling in the y dimension) and the B-type (traveling in the x dimension) messages can coexist within the first two while loops, we can exploit communication links of 2D mesh as much as possible. Unlike the previous versions, groups of pixel message are sent asynchronously and are not delayed until the end; therefore, shorter messages are needed to be flushed (i.e., network congestion can be less).

We do not leave the first while loop until all local triangles are rendered. After this stage, we flush each A-type message with a flag indicating it is the last A-type message from a sender to other receivers. Similarly, we flush all B-type messages with flags after the second while loop. To exit the second and third while loops, we must guarantee that all last A-type (exits from the second) and B-type (exits from the third) messages have arrived from other senders. This arrangement ensures that the third loop can start only after the exit from the second loop. It means that each processor must completely finish its image composition in the y dimension in the second loop before ending the composition in the x dimension. There is no A-type message arriving in the third loop, but both the A-type and the B-type messages can arrive in the first two loops. In all, each processor needs to receive $r - 1$ number of the A-type last messages and $c - 1$ number of the B-type last messages, respectively.

5 Performance Evaluation and Experimental Results

To perform our experiments, we used five datasets from Eric Haines's SPD database [30]. Table 2 shows the sizes of the different datasets in our tests. These datasets represent different object distributions in the image screen with sizes ranging from 150K triangles to 500K triangles. Figs. 9, 10, 11, 12, 13 show the rendering results for the five scenes. In our implementation, each large group consists of 2,000 triangles and the data for each triangle is 48 bytes. The rendering rate measured here does not count the time neither to save computed pixels nor to reconstruct the final image and to input data from disk. Similar measurements were performed in [9], [12].

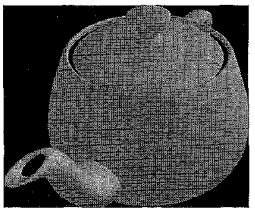


Fig. 9. Teapot.

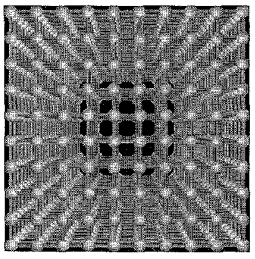


Fig. 10. Lattice.

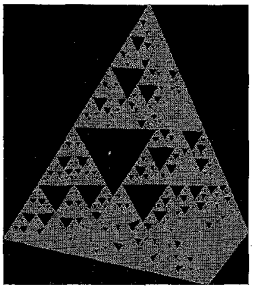


Fig. 11. Tetra.

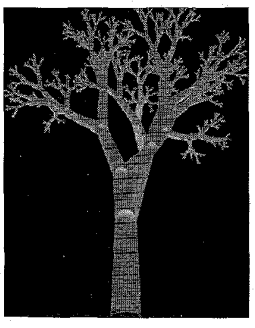


Fig. 12. Tree.

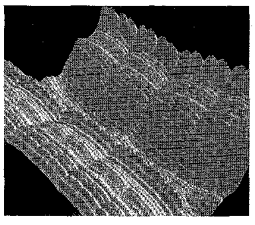


Fig. 13. Mountain.

TABLE 2
NUMBER OF TRIANGLES AND DATA SIZE OF THE FIVE TEST SCENES

Scene	Number of Triangles	Size of Dataset
Mountain	524,288 (512K)	24.0 Mbytes
Tree	425,776 (416K)	19.5 Mbytes
Tetra	262,144 (256K)	12.0 Mbytes
Lattice	235,200 (230K)	10.7 Mbytes
Teapot	159,600 (155K)	7.3 Mbytes

TABLE 3
SOME SCENE STATISTICS OF FIVE TEST SCENES IN 20 ZOOM-IN SEQUENCES

Average Triangle Size	Subpixel to 3.5 Pixels
Scene	Depth Complexity
Mountain	0.35 to 2.15
Tree	0.04 to 1.30
Tetra	0.12 to 1.01
Lattice	0.14 to 1.50
Teapot	0.08 to 0.90

These five scenes were run on different number of processors, rendering the image screen at 512 × 512 resolution without anti-aliasing. Each scene was illuminated by a single light source and shaded by Gouraud shading. The reported timing was obtained by averaging the rendering times for 20 frames. We controlled the viewpoint to allow "zoom-in" effect in these 20 frames. The purpose of this is to represent a more general image distribution. In our evaluation, we start by rendering an image whose objects are projected close to the center of the screen, and continuously zoom-in until objects show in most areas of the screen. The different image distributions give a fair comparison study. Similar measurements were performed in [12]. In addition, some scene statistics of five test scenes in these 20 frames is given in Table 3, and the Appendix shows some of the "zoom-in" sequences to show what was actually rendered ("Teapot" scene, for example). In Table 3, the range in depth complexity is for the scenes as the view was zoomed. The triangle sizes for the "zoom-in" sequences of each scene range from subpixel size to 3.5 pixels on the average. Note that we handle subpixel sized triangles by discarding them in our implementation.

Fig. 14 and Fig. 15 show the rendering rates of the PP scheme and the PPB scheme. The rendering rate increases as more processors are used. Using 512 processors to render five test scenes, the PP scheme achieves 0.25 – 0.8 million triangles/sec. On the other hand, the PPB scheme achieves 0.5 – 1.3 triangles/sec. Without bounding box optimization, the rendering rate of the PP scheme is slowed down by 20% – 50%, due to the differing distributions of sparse pixels in different scenes. Therefore, it is very important to take advantage of sparse pixels to achieve high rendering rate. Using bounding box optimization, many pixels are deleted at earlier stages. Thus, we get a better rendering rate. Using 16 × 32 processors, the PPB peak performance is about 1.3 million triangles/sec for the "Tree" scene while our lowest performance is about 500 K triangles/sec for the "Teapot" scene. The rendering rate for the other scenes is about 1 million triangles/sec. We note that additional costs described previously in PPB scheme are less significant compared with performance improvement obtained by the use of bounding box.

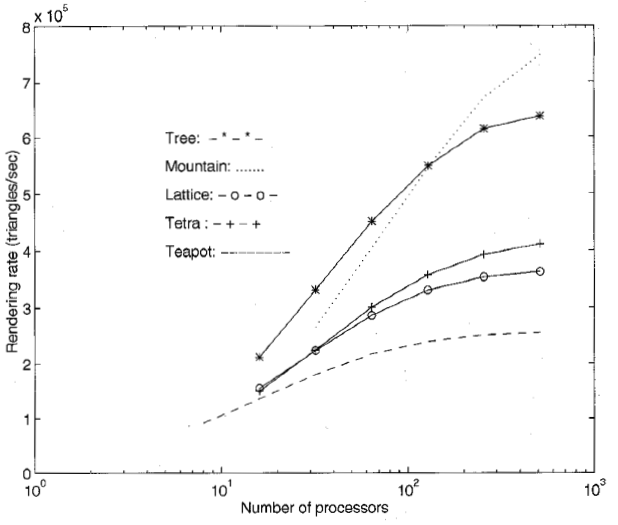


Fig. 14. The rendering rates for five test scenes using different numbers of processors and the PP scheme.

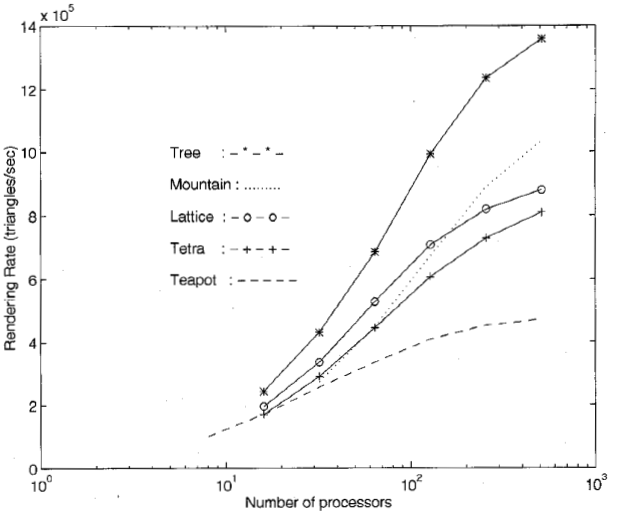


Fig. 15. The PPB rendering rate for five test scenes using different numbers of processors.

Fig. 16 and Fig. 17 show rendering rates for the other two versions: DPF and DPFL. For the DPFL, the composition speed is also slightly improved by the static load balancing and thus yields better rendering rate. Using 512 processors for five test scenes, the DPF scheme achieves 2.5 – 4.0 million triangles/sec and the DPFL scheme achieves 2.8 – 4.0 million triangles/sec respectively. These two performance graphs (Fig. 16 and Fig. 17) show that our performance is much better than those of earlier versions. This superior performance is obtained from significant improvement in the composition speed. Fig. 17 shows that the performance does not drop off for up to 512 processors. However, we can expect the performance to begin to decline beyond 512 processors. As the rendering time approaches zero, the total time is dominated by the composition time, which will gradually increase with increase in number of processors and finally slow down the overall

rendering rate. Furthermore, since the composition time remains constant, our rendering rate would be better for the same size image if we had a larger number of triangles: there would be more rendering computation, with the same composition cost. The rendering rates for the "Tree" and the "Mountain" scenes are better than those of the other three scenes. In the "Tree" scene, we achieve high peak performance due to large number of "inactive" pixels. As mentioned earlier, there is some similarity between DPF scheme and Ellsworth's work [12]. He achieved a peak rendering rate of about 2.8 million triangles per second on a 806,640 triangle dataset using 256 processors. In most cases, the resulting performance on his test scenes begins to decline after 128 or 256 processors. For both DPF and DPFL schemes, the rendering rate do not decline for up to 512 processors on our test scenes.

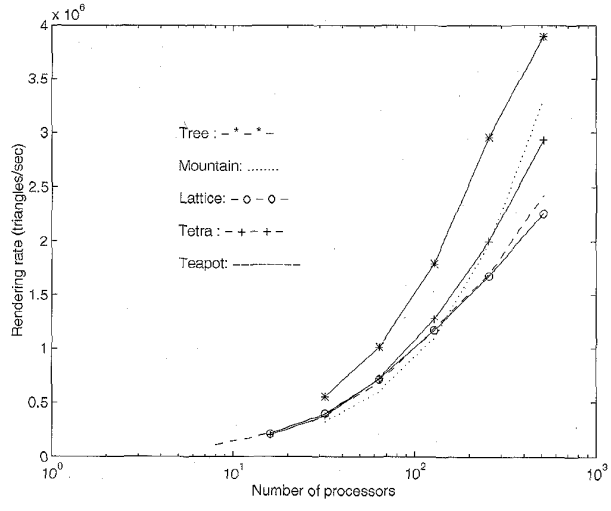


Fig. 16. The rendering rate for five test scenes using different numbers of processors and the DPF scheme.

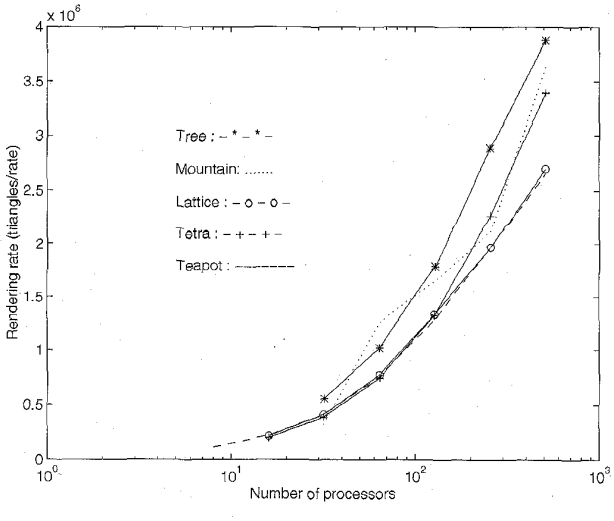


Fig. 17. The rendering rate for five test scenes using different numbers of processors and the DPFL scheme.

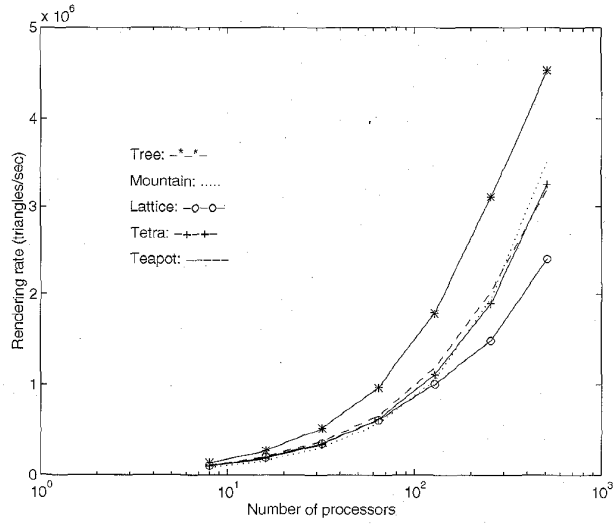


Fig. 18. The rendering rates for five test scenes using different numbers of processors and the DPFS scheme.

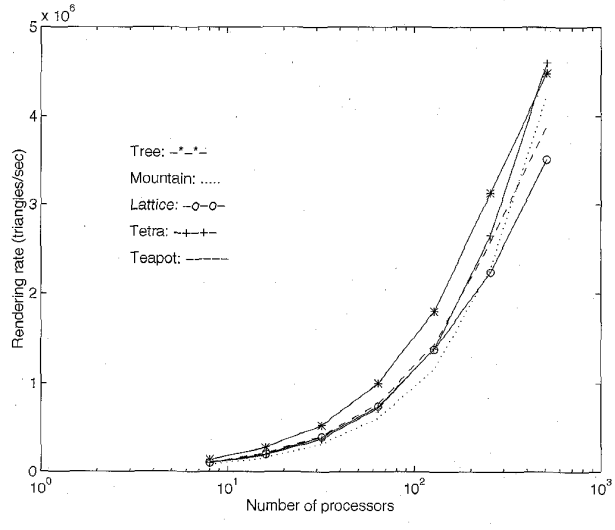


Fig. 19. The rendering rates for five test scenes using different numbers of processors and the DPFSL scheme.

For the DPFS implementation, the sizes of both the *A*-type and the *B*-type message buffers are fixed at 4K bytes after several tests on five test scenes to tradeoff the excessive memory usage and message latency. In the first while loop, we need to switch between the rendering pipeline and message handling when a message arrives. After some number of n triangles are rendered, we check to see if any *A* or *B*-type message has arrived from other processors, using msgdone() on the Delta. Each msgdone() can add extra 2μ seconds to the overall cost on the Delta. The question arises as to what should be the appropriate value of n. Our experiments show that it does not make much difference after a certain fixed value. The main concern in choosing n is to avoid very small values, in order to reduce the overhead generated from lots of calls to msgdone(), which checks for incoming pixel message. In our current implementation, we

TABLE 4
COMPARISON BETWEEN THE DPF AND THE DPFS SCHEMES, AND THE DPFL AND THE DPFSL SCHEMES FOR THE "TEAPOT" SCENE

#procs	8	16	32	64	128	256	512
DPF	1.461	0.778	0.424	0.241	0.150	0.118	0.118
DPFS	1.489	0.785	0.428	0.245	0.137	0.078	0.050
Gain	−1.916%	−0.900%	−0.943%	−1.660%	8.667%	33.898%	57.627%
DPFL	1.386	0.716	0.384	0.211	0.124	0.081	0.059
DPFSL	1.460	0.749	0.395	0.207	0.111	0.062	0.041
Gain	−5.339%	−4.609%	−2.865%	1.896%	10.484%	23.457%	30.508%

TABLE 5
THE COMPARISON IN RENDERING RATE BETWEEN THE DPFL AND THE DPFSL SCHEMES FOR FIVE TEST SCENES

#procs	8	16	32	64	128	256	512
Teapot							
DPFL	115,152	222,905	415,625	756,398	1,287,097	1,970,370	2,705,085
DPFSL	109,315	213,084	404,051	771,014	1,437,838	2,574,194	3,892,683
Lattice							
DPFL	—	216,176	412,632	776,238	1,344,000	1,993,220	2,734,884
DPFSL	104,289	200,341	392,000	739,623	1,383,529	2,240,000	3,510,448
Tetra							
DPFL	—	200,263	388,938	744,727	1,337,469	2,279,513	3,404,468
DPFSL	96,376	189,001	373,425	722,160	1,387,005	2,647,919	4,599,018
Tree							
DPFL	—	—	555,118	1,025,966	1,796,523	2,898,407	3,870,691
DPFSL	138,921	275,842	528,258	1,001,826	1,804,136	3,130,706	4,481,853
Mountain							
DPFL	—	—	322,638	585,142	1,134,823	2,114,065	3,666,350
DPFSL	80,021	158,156	314,321	607,518	1,162,501	2,279,513	4,228,129

used a value of 30 for n. In the future, we would like to develop some performance model like Crockett and Orloff's work [9] and base on it to automatically determine the size of buffers and the value of n. We implement another variant of DPFS scheme termed DPFSL with static load balancing by assigning horizontal lines among the processors in an interleaved fashion. Fig. 18 and Fig. 19 show rendering rates for both DPFS and DPFSL. Both DPFS and DPFSL schemes achieve the best rendering rate on "Tree" scene at about 4.5 million triangles per second using 512 processors.

Table 4 shows the total rendering time comparison between DPF and DPFS, and DPFL and DPFSL for the "Teapot" scene. For large systems, DPFS outperforms DPF, and DPFSL outperforms DPFL significantly. For example, using 512 processors, DPFSL achieves performance gain by about 31% over DPFL. While load balancing is quite important for better composition performance, we see that DPFS scheme performs better than DPFL using both 256 and 512 processors. In large systems, communication overhead is more dominant on performance than pixel composition computation. On the other hand, for small systems, both DPFS and DPFSL are slightly slowed down (at most 5%) for the "Teapot" scene using eight processors. This is due to extra overheads incurred in three while loops. These include buffer management, message detection, breaking of rendering pipeline by inserting message handling code in the first while loop and so on. In the case of small systems, the saving of message communication time cannot offset these factors and results in slight slow-down in performance. Table 5 shows similar behavior in rendering rate comparison. Therefore, our results indicate that task scheduling between communication and rendering work is quite important to achieve better performance on large systems.

Table 5 shows that DPFSL consistently performs better than DPFL for large systems. DPFSL achieves a rendering rate of 3.5 to 4.6 million triangles/sec using 512 processors. In comparison with DPFL, we gain one half to one million triangles/sec in rendering rate. For example, replacing DPFL with DPFSL, the performance of rendering "Tetra" scene changes from 3.4 to 4.6 million triangles/sec (i.e., performance gain of 13% to 30%). Surprisingly, unlike earlier versions, the performance of five test scenes do not vary significantly. Again, the rendering performance do not drop off for up to 512 processors in both DPFS and DPFSL schemes.

Tables 6 and 7 show the time breakdown of our renderer for the "Teapot" and "Tree" scenes using four different composition schemes. We divide the total rendering time into two main parts: rendering time (Rend) that consists of exact rendering time and pre-processing time for composition, and the composition (Comp) time. From both tables, we see that the rendering time decreases slightly linearly as the number of processors are increased for both scenes. Among these six versions, PP scheme needs least rendering time, since it needs less preprocessing time for composition. In each version of composition (except DPFS and DPFSL), the composition times for both scenes are almost constant, regardless of the number of processors. For both DPFS and DPFSL, the overlap between communication and rendering work causes the difference in composition time using different number of processors.

Table 8 shows the speedup and efficiency for DPFSL scheme. Both values are based on the times obtained from the minimum configuration that test scene can fit in. For example, it needs eight processors to hold all test scenes in DPFSL implementation. The DPFSL scheme scales well first

TABLE 6
THE TIME (IN SECONDS) FOR THE "TEAPOT" SCENE

#procs	8	16	32	64	128	256	512
PP							
Rend	1.214	0.601	0.306	0.154	0.088	0.055	0.038
Comp	0.534	0.574	0.578	0.579	0.578	0.580	0.587
Total	1.748	0.175	0.884	0.733	0.666	0.635	0.625
PPB							
Rend	1.395	0.705	0.367	0.196	0.107	0.066	0.045
Comp	0.186	0.228	0.254	0.279	0.282	0.286	0.293
Total	1.581	0.933	0.621	0.475	0.389	0.352	0.338
DPF							
Rend	1.360	0.691	0.354	0.179	0.092	0.049	0.024
Comp	0.101	0.087	0.070	0.062	0.058	0.069	0.094
Total	1.461	0.778	0.424	0.241	0.150	0.118	0.118
DPFL							
Rend	1.325	0.674	0.347	0.176	0.090	0.048	0.024
Comp	0.061	0.042	0.037	0.035	0.034	0.033	0.035
Total	1.386	0.716	0.384	0.211	0.124	0.081	0.059
DPFS							
Rend	1.381	0.697	0.364	0.180	0.093	0.053	0.027
Comp	0.108	0.088	0.064	0.065	0.044	0.025	0.023
Total	1.489	0.785	0.428	0.245	0.137	0.078	0.050
DPFSL							
Rend	1.345	0.670	0.351	0.182	0.093	0.050	0.026
Comp	0.115	0.079	0.044	0.025	0.018	0.012	0.015
Total	1.460	0.749	0.395	0.207	0.111	0.062	0.041

TABLE 7
THE TIME (IN SECONDS) FOR THE "TREE" SCENE

#procs	16	32	64	128	256	512
PP						
Rend	1.433	0.710	0.368	0.198	0.113	0.085
Comp	0.573	0.575	0.576	0.578	0.580	0.583
Total	2.006	1.285	0.944	0.776	0.693	0.668
PPB						
Rend	1.588	0.812	0.416	0.221	0.120	0.061
Comp	0.161	0.173	0.205	0.208	0.224	0.252
Total	1.749	0.985	0.621	0.429	0.344	0.313
DPF						
Rend	1.501	0.734	0.395	0.207	0.111	0.072
Comp	0.056	0.054	0.051	0.054	0.059	0.083
Total	1.557	0.788	0.446	0.261	0.170	0.155
DPFL						
Rend	1.502	0.734	0.383	0.206	0.114	0.073
Comp	0.034	0.033	0.032	0.031	0.032	0.035
Total	1.536	0.767	0.415	0.237	0.146	0.108
DPFS						
Rend	1.512	0.742	0.408	0.217	0.118	0.080
Comp	0.089	0.079	0.034	0.021	0.019	0.014
Total	1.601	0.821	0.442	0.238	0.137	0.094
DPFSL						
Rend	1.501	0.738	0.391	0.211	0.118	0.081
Comp	0.079	0.068	0.034	0.025	0.019	0.014
Total	1.580	0.806	0.425	0.236	0.137	0.095

with about 98% efficiency, but decreases to about 50% ("Tree" scene) at the largest configuration of the Delta. As the size of the machine increases, one of the reasons for decreasing efficiency is that rendering load becomes uneven among the processors when fewer triangles are computed on each processor. The major cause is that composition time does not scale with increasing number of processors. With the increase in the number of processors, the composition time of each scene is almost kept constant while the rendering time decreases, and when composition time becomes dominant or comparable to rendering time, the efficiency will decrease.

Table 9 shows the effects of the group interleaving data distribution on rendering performance using PPB scheme. We experimentally compared rendering rates using interleaving per-primitive and group interleaving data distribution and our results show that the latter can lead to better rendering performance for the PPB scheme. For SPD database, the sizes of triangles contained in each scene are quite similar, and thus group interleaving does not incur too much load imbalances in rendering part as simple interleaving does. However, group interleaving tends to result in tighter bounding boxes for the sort-last-full system as discussed in an earlier section. Therefore, for our test scenes,

TABLE 8
THE SPEEDUP AND EFFICIENCY DATA USING THE DPFSL SCHEME

#procs	8	16	32	64	128	256	512
Speedup							
Teapot	1.00	1.95	3.70	7.05	13.15	23.54	35.61
Lattice	1.00	1.92	3.76	7.09	13.27	21.48	33.66
Tetra	1.00	1.96	3.87	7.49	14.39	27.47	47.71
Tree	1.00	1.99	3.80	7.21	12.99	22.54	32.26
Mountain	1.00	1.98	3.93	7.59	14.53	28.49	52.84
Efficiency							
Teapot	100%	98%	93%	88%	82%	74%	56%
Lattice	100%	96%	94%	89%	83%	67%	53%
Tetra	100%	98%	97%	94%	90%	86%	75%
Tree	100%	100%	95%	90%	81%	70%	50%
Mountain	100%	99%	98 %	95%	91%	89%	83%

TABLE 9
THE RENDERING RATE FOR THE "LATTICE" AND "TETRA" SCENES USING INTERLEAVING (I) AND GROUP INTERLEAVING (GI)

#procs	16	32	64	128	256	512
Lattice						
Int.	146,605	263,367	428,025	614,500	768,501	878,431
Group Int.	195,454	336,000	528,361	707,687	818,798	879,745
Tetra						
Int.	106,271	189,828	314,415	463,233	641,252	711,574
Group Int.	170,511	290,496	445,974	606,463	728,076	809,336

group interleaving works better than per-primitive interleaving. The group interleaving technique needs to be investigated further. For example, we can address questions like what's the optimal size for a group? How does this behavior depend on the ordering of primitives in the input data set?

As mentioned earlier in this section, we do not count the time to reconstruct the final image in our rendering rate measurement as was done in [9], [12]. In the case that we need to assemble image fragments scattered among the processors into a finished image, we achieve this by performing DPF-like scheme (message sending only but no message composition) in reverse order. Take Fig. 7, for example, in the end of composition stage, processor P_i will hold $Z_{(i+1) mod N}$. As the image gathering starts, P_1, P_2, and P_3 all send their portion of subimage to P_0. For 2D mesh architecture, we perform similar routine in the x dimension first and then in the y dimension. The cost of gathering 512 × 512 resolution image using different number of processors is given in Table 10. The cost of image reconstruction varies from 0.051 to 0.082 seconds for different number of processors. For small systems, this overhead will not affect the overall rendering rate too much (less than 10% on five test scenes). But for large systems, it can even slowdown the rendering rate by about a factor of two (take "Teapot" scene for example).

TABLE 10
THE IMAGE RECONSTRUCTION TIMINGS (SECONDS) AT 512 × 512 RESOLUTION USING DIFFERENT NUMBER OF PROCESSORS

#procs	16	32	64	128	256	512
Gathering	0.051	0.054	0.062	0.074	0.080	0.082

6 CONCLUSIONS AND FUTURE WORK

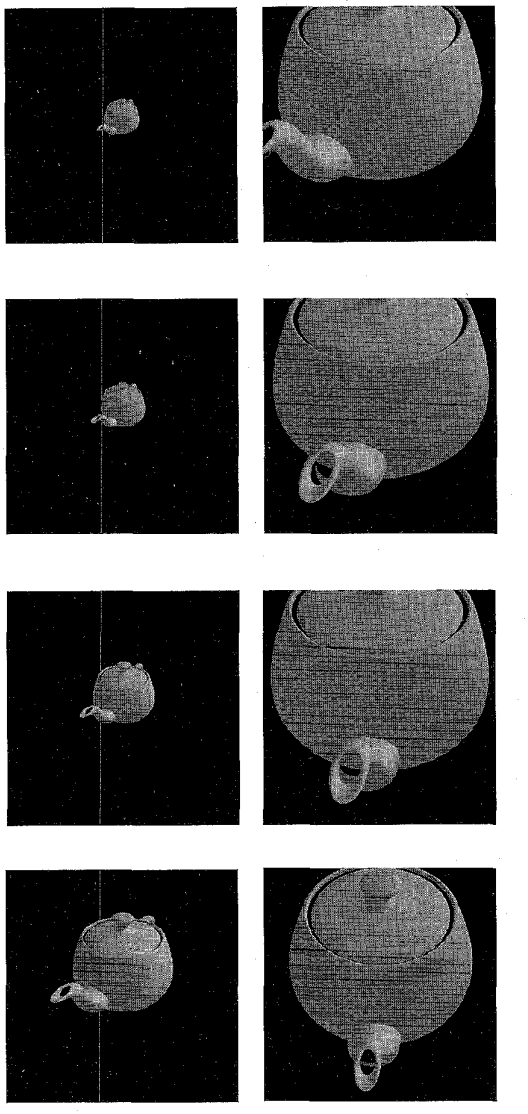
In this paper, a sort-last parallel polygon rendering system is described for use on a 2D mesh connected multicomputer system. We implemented a family of image composition schemes for the sort-last rendering system. We start from a sort-last-full image composition scheme, termed PP, and present many variants of it to provide very fast pixel composition. All proposed schemes consist of two phases on a 2D mesh architecture (i.e., $r \times c$ processors). In the first phase, our schemes are executed along the y dimension which needs $r - 1$ steps. It takes another $c - 1$ steps along the x dimension in the second phase. These alternatives of the PP include bounding box optimization (sort-last-full, PPB)), sort-last-sparse (coherent regions and interleaved composition regions, DPF and DPFL)) and task scheduling between communication and rendering (DPFS and DPFSL).

We experimentally compared all proposed methods on Caltech's Intel Delta, a 512 processor multicomputer system. The exceptionally superior performance of the DPFS and the DPFSL schemes provides evidence that sort-last-sparse strategies are better suited for software implementation on general purpose multiprocessor systems. Our experimental results show that schemes based on interleaved composition regions perform better schemes with coherent regions. In large systems, scheduling the tasks of rendering and communication can improve the sort-last-sparse schemes significantly while incurring small overheads in buffer management. We also evaluated a well-known binary-swap composition scheme (sort-last-full) and showed that binary-swap could not perform better than the PP (sort-last-full) on the 2D mesh Intel Delta. We used five public domain datasets to evaluate our implementation. With 512 × 512 resolution image, our final version, DPFSL, achieved the peak performance close to 4.6 million triangles per second which is higher than any other multicomputer implementation known to the authors.

There is scope for further work in several directions. First, we plan to accommodate features such as antialiasing and transparency. Second, we will find suitable load balancing scheme to even out load imbalances incurred in the rendering part. In sort-last class schemes, load balancing can become an important issue as the sizes of polygons have large disparities. Such datasets are common in scientific applications with nonuniform grids: grid cell

sizes may vary over several orders of magnitude, and polygons derived from these grid cells exhibit similar variations in size. Third, our composition scheme can be exploited for parallel volume rendering problem and will be experimentally evaluated in the near future.

APPENDIX



ACKNOWLEDGMENTS

This research was performed in part using the Intel Touchstone Delta System operated by California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest Laboratory (PNL), a multiprogram laboratory operated for the U.S. Department of Energy by Battelle Memorial Institute under Contract DE-AC06-76RLO 1830. We would also like to give our special thanks to anonymous reviewers for useful comments on our composition scheme. This research is supported by the Boeing Centennial Chair Professor funds.

REFERENCES

[1] T.-Y. Lee, C.S. Raghavendra, and J.B. Nicholas, "Image Composition Methods for Sort-Last Polygon Rendering on 2D Mesh Architectures," *Proc. 1995 Parallel Rendering Symp.*, ACM, pp. 55-62, Oct. 1995.

[2] S. Molnar and H. Fuchs, "Advanced Raster Graphics Architecture," *Computer Graphics: Principles and Practice*, second edition, J.D. Foley et al., eds., pp. 855-923 Reading, Mass.: Addison-Wesley, 1990.

[3] T.W. Crockett, "Design Considerations for Parallel Graphics Libraries," *Proc. Intel Supercomputer Users Group*, pp. 3-14, June 1994.

[4] S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*. Wellesley, Mass.: AK Peters, Ltd., 1992.

[5] J. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics," *Computer Graphics*, vol. 16, no. 3, pp. 127-133, July 1982.

[6] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 23-32, July 1994.

[7] C. Mueller, "The Sort-First Rendering Architecture for High Performance Graphics," *Proc. 1995 Symp. Interactive 3D Graphics*, pp. 75-84, ACM SIGGRAPH, Apr. 1995.

[8] M. Cox, "Algorithms for Parallel Rendering," PhD dissertation, Dept. of Computer Science, Princeton Univ., May 1995.

[9] T.W. Crockett and T. Orloff, "Parallel Polygon Rendering for Message-Passing Architectures," *IEEE Parallel and Distributed Technology*, pp. 17-28, Summer 1994.

[10] S. Whitman, "Dynamic Load Balancing for Parallel Polygon Rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 41-48, July 1994.

[11] D. Roble, "A Load-Balanced Parallel Scanline Z-buffer Algorithm for the iPSC Hypercube," *Proc. First Int'l Conf. Pixim 88*, pp. 177-192, Editions Hermes, Paris, 1988.

[12] D. Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 33-40, July 1994.

[13] Kubota Pacific Computer, Denali Technical Overview, version 1.0, Mar. 1993.

[14] Evans and Sutherland Computer Corporation, Freedom Series Technical Report, Oct. 1992.

[15] S. Molnar, "Image-Composition Architectures for Real-time Image Generation," PhD dissertation, Univ. of North Carolina at Chapel Hill, Oct. 1991.

[16] D. Fussel and B.D. Rathi, "A VLSI-Oriented Architecture for Real-time Raster Display of Shaded Polygons," *Proc. Graphics Interface '82*, pp. 373-380.

[17] C.D. Shaw, "A VLSI Architecture for Image Composition," *Advanced in Graphics Hardware III*, pp. 183-199, Springer-Verlag, 1988.

[18] R. Heiland, "Object-Oriented Parallel Polygon Rendering," *Proc. Gviz '94*, pp. 19-26, Richland, Wash., Tri-Cities ACM-SIGGRAPH chapter, Sept. 1994.

[19] J. Li and S. Miguet, "Z-buffer on a Transputer-Based Machine," *Proc. Sixth Distributed Memory Computing Conf.*, pp. 315-322, IEEE, 1991.

[20] M. Cox and P. Hanrahan, "A Distributed Snooping Algorithm for Pixel Merging," *IEEE Parallel and Distributed Technology*, pp. 30-36, Summer 1994.

[21] K. Ma, J.S. Painter, and M.F. Krogh, "Parallel Volume Rendering Using Binary Swap Composition," *IEEE Computer Graphics and Application*, vol. 14, no. 4, pp. 59-67, July 1994.

[22] R. J. Karia, "Load Balancing of Parallel Volume Rendering with Scattered Decomposition," *Proc. Scalable High Performance Computing Conf.*, May 1994.

[23] C.M. Wittenbrink, "Designing Optimal Parallel Volume Rendering Algorithms," PhD dissertation, Univ. of Washington, June 1993.

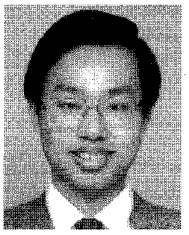
[24] C.M. Wittenbrink and M. Harrington, "A Scalable MIMD Volume Rendering Algorithms," *Proc. Eighth Int'l Parallel Processing Symp.*, pp. 916-920, Cancun, Mexico, Apr. 1994.

[25] P. Li and D.W. Curkendall, "Parallel Three Dimensional Perspective Rendering," *Proc. Second European Workshop Parallel Computing*, pp. 320-331, Mar. 1992.

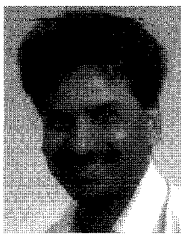
[26] P. Li, W.H. Duquette, and D.W. Curkendall, "Remote Interactive Visualization and Analysis (RIVA) Using Parallel Supercomputers," *Proc. 1995 Parallel Rendering Symp.*, pp. 71-78, ACM, Oct. 1995.

[27] F. Ortega, C. Hansen, and J. Ahrens, "Fast Data-Parallel Polygon Rendering," *Proc. Supercomputing '93*, pp. 709-78, IEEE, 1993.

[28] C.D. Hansen, M. Krogh, and W. White, "Massively Parallel Visualization: Parallel Rendering," *Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computing*, pp. 790-795, SIAM, Feb. 1995.
[29] M. Barnett, R. Littlefield, D.G. Payne, and R. van de Geijn, "Global Combine on Mesh Architectures with Wormhole Routing," *Proc. Seventh Int'l Parallel Processing Symp.*, Apr. 1993.
[30] E. Haines, "A Proposal for Standard Graphics Environments," *IEEE Computer Graphics and Application*, pp. 3-5, July 1987.
[31] R. Littlefield, "Characterizing and Tuning Communications Performance on the Touchstone Delta and iPSC/860," *Proc. 1992 Ann. Conf.* Intel Supercomputer Users Group, pp. 309-313, Oct. 1992.
[32] Y.C. Tay, "A Performance Analysis of Object-Parallel Rendering and Composition," Research Report no. 569, Dept. of Mathematics, National Univ. of Singapore, May 1993.



Tong-Yee Lee received his BS in computer engineering from Tatung Institute of Technology in Taipei, Taiwan, in 1988, his MS in computer engineering from National Taiwan University in 1990, and his PhD in computer engineering from Washington State University, Pullman, in May 1995. He is an associate professor in the Department of Information Management at Nantai College, Tainan County, Taiwan, Republic of China, Prior to joining the PhD program at Washington State University in 1992, he worked for Hitron Technology in 1990 and Tatung in 1991, both in Taipei, Taiwan. He has collaborated with the Institute of Computer and Information Engineering at national Sun Yat-Sen University in Koushung, Taiwan. He leads a team in distributed systems and computer graphics at Nantai College. His research interests include parallel rendering design, computer graphics, visualization, virtual reality, parallel processing, distributed systems, multimedia networking, heterogeneous computing, and collaborative framework design.



C.S. Raghavendra received the BSc (Hons) physics degree from Bangalore University in 1973 and the BE and ME degrees in electronics and communication from the Indian Institute of Science, Bangalore, in 1976 and 1978, respectively. He received the PhD degree in computer science from the University of California at Los Angeles in 1982. From September 1982 to December 1991, he was a member of the faculty of the Electrical Engineering-Systems Department at the University of Southern California, Los Angeles. He is currently the Boeing Centennial Chair Professor of Computer Engineering at the School of Electrical Engineering and Computer Science at Washington State University, Pullman. His research interests are high speed networks, parallel processing, fault tolerant computing, and distributed systems. Dr. Raghavendra was a recipient of the Presidential Young Investigator Award in 1985. He is a senior member of the IEEE.

John B. Nicholas turned to the sciences after a career as a professional musician. He received his BS in biochemistry from Illinois Benedictive College in 1986 and his PhD in physical chemistry from the University of Illinois at Chicago under the direction of Anton J. Hopfinger. He is currently a senior research scientist at Pacific Northwest Laboratory, where his research focuses on the theoretical study of heterogeneous catalysis and high-performance computations chemistry methodology development.

Dr. Nicholas is the author or coauthor of 30 papers and has made numerous presentations at national and international meetings. He is a member of the American Chemical Society and the American Geophysical Union.