OBJECT-SPACE PARALLEL POLYGON RENDERING ON HYPERCUBE-CONNECTED MULTICOMPUTERS

Tahsin M. Kurç, Cevdet Aykanat and Bülent Özgüç

BILKENT UNIVERSITY

Department of Computer Engineering and Information Science

Technical Report BU-CEIS-94-24

This work is partially supported by Intel Supercomputer Systems Division grant no. SSD100791-2 and Turkish Scientific and Technical Research Council (TÜBİTAK) grant no. EEEAG-5

Object-Space Parallel Polygon Rendering on Hypercube-connected Multicomputers[1]

Tahsin M. Kurç, Cevdet Aykanat and **Bülent Özgüç**

Department of Computer Engineering and Information Sciences
Bilkent University, Ankara, TURKEY.

Abstract

This paper presents algorithms developed for object-space parallelism for polygon rendering on hypercube-connected multicomputers. In object-space parallelism, each processor is given the responsibility to render a portion of the input polygons. Each processor shades and z-buffers the locally generated pixels. After this local rendering, the remaining pixels in each processor should be globally z-buffered. Efficient parallelization of this pixel merging operation is important. This is the only place where an overhead is incurred due to parallelization. In this paper, efficient algorithms are presented to perform local rendering and global pixel merging operations. A modified scanline based z-buffer algorithm is proposed. This algorithm reduces the number of pixels sent and received in the pixel merging operation and avoids the re-initialization of the z-buffer for each scanline. Various algorithms are proposed for pixel merging phase. These algorithms use different communication characteristics of the hypercube multicomputers. Efficient algorithms for load balancing in the pixel merging operation is also proposed and presented. Experimental results obtained on a 16-processor Intel's iPSC/2 hypercube multicomputer are also presented.

1 Introduction

Fast image generation on computers has been a challenge for many years. A lot of efforts have been put to develop faster algorithms to generate images on computers in real-time or near real-time. However, along with the advances in image generation techniques, increased importance of more realism in computer generated images has made the rendering process more and more complex and time consuming. In addition, increased complexity of image databases (e.g. large number of polygons that make up the scene) has required more and more memory space. Hence, efforts have been put to build special purpose graphics hardware to render more complex scenes in affordable times. However, one major drawback of special purpose graphics hardware is their limited flexibility. Only hardwired algorithms can be used in such graphics machines and it is very difficult to incorporate different techniques to such

[1]This work is partially supported by Intel Supercomputer Systems Division grant no. SSD100791-2 and Turkish Scientific and Technical Research Council (TÜBİTAK) grant no. EEEAG-5

architectures. General purpose parallel computers, on the other hand, can provide a cost-effective and flexible environment for fast image generation. Effective utilization of parallel processing in computer graphics requires the subdivision and mapping of data and computations to the processors. There are two approaches, in particular, for the subdivision of data and computations; image-space parallelism and object-space parallelism. In the image-space parallelism [3, 4, 5], image-space is divided among the processors of the parallel computer. Each processor is responsible for rendering its local portion of the screen. In object-space parallelism [6, 7], the objects or polygons that make up the environment are partitioned among the processors. Each processor is responsible for rendering its local portion of the object database.

This paper investigates the object-space parallelism on hypercube connected distributed memory multicomputers. A d-dimensional hypercube consists of $P = 2^d$ processors (nodes) with a link between every pair of processors whose binary addresses differ in one bit. Thus, each processor is connected to d other processors. The *channel i* refers to the communication links between processors whose processor ids differ in only the i^{th} bit. The hypercube multicomputer has a recursive structure. That is, two equal (d-1) dimensional hypercubes, or subcubes, can be generated by dividing the hypercube along *channel i.* If a d-dimensional hypercube is divided into two subcubes along *channel (d−1)*, then nodes, whose ids in binary representation are $1xx...x$, are said to be in the upper $(d-1)$ dimensional hypercube. Similarly, nodes, whose ids are $0xx...x$, are said to be in the lower $(d-1)$ dimensional hypercube, where x is either 0 or 1.

2 Terms and Definitions

This section presents some terms and definitions used in this paper.

P : number of processors in the hypercube.

d : dimension of the hypercube, where d = $log_2(P)$.

z : distance of the generated pixel to the screen.

N : number of scanlines on the screen (assumed to be a multiple of P).

A : total number of pixels on the screen where A = NxN, i.e. a square screen is assumed.

Definition 1: A location (x,y) on the image plane is said to be *active*, if at least one pixel is generated for that location. Note that, different processors may generate pixels for the same location.

Definition 2: A pixel is said to be a *winning* pixel, if it is the current pixel whose z value is minimum for the active pixel location. At the end of the pixel merging operation there remains only one winning pixel for each active pixel location.

1. Receive the local polygon information from the host processor.
2. Perform viewing transformations.
3. Reject the back facing polygons, and clip the polygons to the viewing volume.
4. Project the clipped polygons to screen coordinates.
5. Perform hidden-surface removal and shading on the local polygon set.
6. Perform pixel merging to obtain the final picture.

Figure 1: Basic steps of object-space parallel rendering.

3 Object-Space Parallel Rendering

In simple terms, rendering is the process of viewing three dimensional shaded objects and scenes generated in the computer. This viewing process is a collection of operations to produce a realistic picture on the computer screen. These operations are applied on the polygons that constitute the scene to be rendered. Coordinate transformations, clipping and projection operations are performed to transform polygon information in three dimensional world coordinates to two dimensional screen coordinates. In order to produce realistic pictures, hidden surface removal and shading operations are performed on the projected polygons. During hidden surface removal, pixels are generated for screen coordinates that are covered by the projected polygon. The z values of the pixels generated for the same active pixel location are compared and pixel closest to the screen is stored into the frame buffer. More information on rendering can be found in [1, 2].

In object-space parallel rendering, input polygons are partitioned among the processors of the multicomputer. Each processor, then, runs a sequential rendering algorithm for its local polygons. In order to obtain the final picture, pixels generated in each processor should be merged, because more than one processor may produce a pixel for the same screen coordinate. This merge operation should be done in minimal time, because this is the only place where an overhead is incurred due to parallelization. The basic steps of the object-space parallel rendering executed in each processor is given in Fig. 1.

At step 1, the host processor distributes polygons to node processors using scattered assignment scheme. In the scattered assignment scheme, adjacent polygons on each surface should be ordered consecutively. Then, successive polygons in the sequence are assigned to the processors in a round-robin fashion. That is, the first polygon is assigned to processor 0, the next to processor 1, etc. When P polygons are assigned, the next polygon is assigned to processor 0 and this process continues. Step 5 of the algorithm is expected to take almost equal amount of computation for adjacent polygons due to similar view volume of adjacent polygons. Hence, scattered assignment is expected to yield good load balance.

At step 5, each processor performs hidden surface removal and shading for its local polygons. Hidden surface removal can be accomplished by z-buffer or scanline z-buffer algorithm. The z-buffer algorithm is simple to implement, but it requires that whole z-buffer be stored in the local memory of each processor. Hence, z-buffer algorithm is not suitable for parallelization on distributed memory architectures. A scanline z-buffer, on the other hand, requires to store only a z-buffer of height one, hence reduces the memory requirements. In this work, a scanline z-buffer algorithm is used to perform local hidden surface removal.

At step 6, each processor exchanges and merges the local z-buffer results to obtain the final picture. The global z-buffering operations during the pixel merge phase can be considered as overhead to the sequential rendering. Furthermore, each global z-buffering operation necessitates interprocessor communication. Hence, efficient implementation of the pixel merging phase is a crucial factor for the performance of object-space parallel rendering.

A straight-forward implementation of steps 5 and 6 can be done by using a conventional scanline z-buffer algorithm to perform local hidden surface removal. Pixel merging phase (step 6) can be performed by exchanging pixel information for all active and inactive screen locations between processors. However, such a scheme will not produce minimal execution time in pixel merging phase, because pixel information for inactive pixel locations are also exchanged. This introduces an unnecessarily large number of global z-buffering operations and hence large volume of interprocessor communication due to inactive pixel locations. This overhead can be reduced by exchanging only winning pixels in each processor. Hence, during the local hidden surface removal step, each processor should store only its local winning pixels to exchange with other processors. In order to do this, a modified scanline z-buffer algorithm is proposed. Note also that, in conventional scanline z-buffer algorithm, the scanline z-buffer should be initialized for each scanline in the screen. The modified scanline z-buffer algorithm avoids the re-initialization of scanline z-buffer for each scanline on the screen.

4 Modified Scanline Z-buffer Algorithm

In this section a modified scanline z-buffer algorithm is presented. Each processor executes the steps of this algorithm on the polygons in its local memory before pixel merging operation. The steps of the algorithm is given below

1. Create a scanline span list for the local polygons.
2. Perform modified scanline z-buffer on the span list entries. Store the winning pixels in Winning Pixel Array (WPA).

When polygons are projected to the screen, some of the scanlines intersect the edges of the projected polygons. For simplicity, we will assume that polygons are convex

polygons. Hence, a scanline can intersect a polygon at most at two edges of the polygon. Therefore, edge intersections occur in pairs. This pair of intersections is also called a *span*. The scanline *span lists* involve a linked list for each scanline which contains the respective polygon spans. Each span is represented by a bucket, which contains the intersection pair and necessary information for z-buffering and shading. The bucket structure can be represented as follows.

BucketStructure {
    x_l , x_r (leftmost and right most x-axis intersections of the
        polygon edges with the scanline)
    z_l , z_r (z values at intersections x_l and x_r)
    I_l , I_r (intensity values at intersections x_l and x_r)
}

At step 1, span lists are constructed by inserting the spans of the projected polygons to the appropriate scanline lists in sorted (increasing) order according to their x_l values. This sorting allows to perform local z-buffering without initializing the scanline array for each scanline on the screen.

At step 2, spans in the scanline lists are processed, in scanline order, for local z-buffering and shading. Two local arrays are used in order to store only winning pixels. First array is called *Winning Pixel Array* (WPA) used to store the winning pixels. Each entry in this array contains location information, z value, and shading information about the respective local winning pixel. Since the z-buffering is done in scanline order, the pixels in the WPA are in scanline order and pixels in a scanline are stored in consecutive locations. Hence, for location information, only x value of the pixel generated for location (x,y) needs to be stored in WPA. Second array, called Scan_Arry of size N, is a modified scanline z-buffer. Each entry of this array contains the index of the current winning pixel for that location in WPA. That is Scan_Arry[x] gives the index of pixel generated for location "x" in WPA. The steps of the modified z-buffer operation are given in Fig. 2. Initially each entry of the Scan_Arry is initialized to zero. WPA_index points to the next available location in WPA. The structure pixel[x] returns the necessary information about pixel generated for location "x" to be stored in WPA.

Note that, in the local z-buffering operation Scan_Arry is initialized only once. If a location "x" in Scan_Arry has a value less than the current value of "Index", it means that location "x" is generated by a span belonging to previous scanlines. Note that, the value of the "Index" is set to "WPA_index" just before the next scanline is processed. Hence, for the next scanline all the values in Scan_Arry will be less than "Index". Therefore, the Scan_Arry does not have to be initialized again. However, due to comparison made with "Index" value, an extra comparison is introduced for each pixel generated. These extra comparison operations can be reduced as follows. The sorted order of spans in the scanline span lists assures that when a span "j" in scanline "i" is rasterized, it will not generate a pixel location "x" which is less than x_l of previous spans. Hence, when a span is processed, first its left intersection point (x_l) in the Scan_Arry, i.e. Scan_Arry[x_l], is compared with "Index". If Scan_Arry[x_l]

```
Initialize Scan_Arry to zero;
Index = 1; WPA_index = 1;
for ( each scanline "i" ) do
   max_active_x = 0;
   for ( each bucket "j" in span list for scanline "i" ) do
      if ( Scan_Arry[x_l] < Index ) then
         for ( x = x_l to x_r ) do /* store pixels without comparison */
            WPA[WPA_index] = pixel[x];
            Scan_Arry[x] = WPA_index;
            WPA_index = WPA_index + 1;
         max_active_x = x_r;
      else
         if ( x_r > max_active_x ) then /* divide the span into two parts */
            x_end = max_active_x;
            max_active_x = x_r;
         else
            x_end = x_r;
         for ( x = x_l to x_end ) do
            k = Scan_Arry[x];
            if ( WPA[k].distance > pixel[x].distance ) then
               WPA[k] = pixel[x];
         for ( x = (x_end+1) to x_r ) do
            WPA[WPA_index] = pixel[x];
            Scan_Arry[x] = WPA_index;
            WPA_index = WPA_index + 1;
   Index = WPA_index;
endfor
```

Figure 2: Modified local z-buffer algorithm.

is less than the current value of "Index", the pixel locations from x = x_l to x = x_r can be updated and pixels generated for those locations are stored to WPA without any distance comparisons. Note that, the variable "max_active_x" holds the maximum "x" location generated by previous spans in the same scanline "i". If Scan_Arry[x_l] is greater than the current value of "Index", this means that a pixel has been generated by a previous span in the current scanline "i". In such a case, the span "j" is divided into two parts at location "max_active_x". For pixels generated for pixel locations less than "max_active_x", distance comparisons are made with the pixels in WPA. The pixels generated for pixel locations greater than "max_active_x" can be stored to WPA without any distance comparisons.

5 Pixel Merging

This section describes parallel pixel merging algorithms. In the hypercube multicomputer, processors can communicate to neighbor and non-neighbor processors. In the

first algorithm for pixel merging only neighbor processors communicate. In the second algorithm, processors send messages to non-neighbor processors as well.

5.1 Pixel Merging by Pairwise Exchange

In this scheme, only neighbor processors communicate with each other. The algorithm is given in Fig. 3. At the first step, the processors in the upper (d-1) dimensional subcube (whose processor ids are $1xx...x$) send their local winning pixels belonging to scanlines in the upper half of the screen (scanlines from 1 to N/2) to neighbor processors in the lower (d-1) dimensional hypercube. Similarly, processors in the lower (d-1) dimensional hypercube (whose processor ids are $0xx...x$) send their local winning pixels belonging to scanlines in the lower half of the screen (scanlines from (N/2+1) to N) to neighbor processors in the upper (d-1) dimensional hypercube. Hence, after step 1, processors in the upper subcube will have the pixels below the half of the screen, and processors in the lower subcube will have the pixels above the half of the screen. Each processor then merges the pixels it receives with the local pixels. The winning pixels after this merge operation is stored into the winning pixel array. Note that, after first exchange step, the hypercube is divided into two subcubes of dimensions d$-$1. Processors in the upper subcube have the pixels belonging to the lower half of the screen. Processors in the lower subcube have pixels belonging to the upper half of the screen. Then each subcube performs, independently form the other subcube, the step 1 on winning pixels in the upper (pixels in processors $0xx...x$) or lower (pixels in processors $1xx...x$) half of the screen. At this step, processors whose ids are $11x...x$ exchanges messages with processors $10x...x$ for lower half of the screen, and similarly processors whose ids are $01x...x$ exchange messages with processors $00x...x$ for the upper half of the screen. This operations are repeated until subcube dimension becomes 0. Hence, at the end of the pairwise exchange and merge operations, each processor has the winning pixels belonging to the N/P portion of the screen. In this scheme number of communication steps is equal to the dimension of the hypercube and only neighbor processors communicate with each other.

This scheme is also called store-and-forward scheme. At each exchange step, the received pixels are stored into the local memory of the processor. These pixels are compared and merged with the pixels stored before. After this merge operation, some part of the winning pixels are sent at the next exchange step, i.e. they are forwarded towards the destination processor through other processors at each concurrent communication step. Note that, during this store-compare-and-forward steps, pixels may be copied from memory of one processor to memory of the other processors more than once. This memory-to-memory copy operations can be reduced by sending the pixels directly to destination processors. The scheme that implements this strategy is given in the next section.

```
m : this node's id (m = b_{d-1}b_{d-2}...b_0 in binary representation).
Start = 1; End = N;
for i = d − 1 to 0 do
    k = m ⊕ 2^i;
    if b_i = 1 then
        Send the winning pixels for scanlines from Start to (Start + End/2 −1) to node k;
        Receive pixels for scanlines from (Start + End/2) to End from node k;
        Start = (Start + End/2);
    else
        Send the winning pixels for scanlines from (Start + End/2) to End to node k;
        Receive pixels for scanlines from Start to (Start + End/2 − 1) from node k;
        End = (Start + End/2 − 1);
    endif

    Merge the pixels received and local pixels, and store the
    winning pixels in winning pixel array;
endfor
```

Figure 3: Pixel merging by pairwise exchange scheme.

5.2 Pixel Merging by All-to-All Personalized Communication

In iPSC/2 hypercube multicomputer communication between processors is done by Direct Connect Modules (DCMs). DCMs allow non-neighboring processors to communicate. Two non-neighbor processors can communicate through the DCMs of the other nodes without store-and-forward overheads and without interrupting the nodes on the route. In iPSC/2 hypercube multicomputer, with DCM technology, communication between two non-neighboring processors is almost as fast as neighbor communications if all the links between two processors is not currently used by other messages. The communication hardware uses the e-cube routing algorithm [9]. Using DCMs, we can exchange messages between non-neighbor processors by the following algorithm [10]. This algorithm ensures that at each exchange step the pixel data is directed to destination processors with the pixel data following disjoint paths.

```
m : this node's id
B_k : the pixel data belonging the partition of screen assigned to processor k.
for i = 1 to P−1 do
    k = m ⊕ i;
    send pixel data B_k to processor k;
    receive pixel data from processor k;
    sync;
endfor
```

In pixel merging by all-to-all personalize communication scheme, the screen is implicitly divided into P equal-sized slices (partitions) each containing N/P scanlines.

Each partition is implicitly assigned to a processor. Then, node "i" sends the pixels belonging to the partition of the processor "k" directly to processor "k". Processors after receiving the pixels wait for the synchronization (*sync*) so that no processor gets ahead of the others and blocks the links to be used by others. This synchronization operation can be executed in $O(log_2 P)$ time. After P−1 exchange steps each processor z-buffers the local pixels and the pixels it receives form other processors. For this, each processor holds a z-buffer of size NxN/P. Local pixels are scattered onto the z-buffer without any distance comparisons. Then, each received pixel's "z" value is compared with the "z" value in the pixel location in the z-buffer. After all the pixels are processed z-buffer contains the winning pixels belonging to the final picture.

This scheme avoids memory-to-memory copy overhead incurred in the previous scheme. However, number of communication steps is increased from $log_2(P)$ to P–1. Hence, for large number of processors with high communication setup time, this scheme may give worse results than pairwise exchange scheme.

6 Load Balancing in Pixel Merging Step

The load balancing in parallel computers is a crucial issue. There are various studies on load balancing [8]. In order to minimize the parallel execution time, an even distribution of work load among the processors of the parallel computer should be achieved. The overhead of the load balancing operation should also be minimized.

In the pixel merging operation, in fact, the screen is implicitly divided among the nodes of the hypercube. In the naive implementations, the screen is divided into P partitions, each of size equal to N/P scanlines. Each partition is implicitly assigned to a processor. However, such a division may result in poor load balancing, because number of active pixel locations may not be the same for each scanline. Hence, during the merge operation some processors may have to merge more pixels than the other processors. This imbalance in the load can be reduced by assigning scanlines to processors adaptively. Two algorithms to implement this adaptive division of screen to processors are given in the following sections.

6.1 Recursive Adaptive Subdivision

This scheme recursively divides the screen into two partitions such that number of pixels in one partition is almost equal to the number of pixels in the other partition. This scheme is well suited to the recursive structure of the hypercube, hence can be done in parallel. The steps of the algorithm are given in Fig. 4.

This algorithm is executed by each processor in the hypercube. The global sum operation at step 2 can be performed in $log_2(P)$ communication steps. At step 6 and 7, if a processor belongs to upper (D−1) dimensional subcube it sets *Start* =

1. Find the amount of winning pixels at each scanline after local z-buffering in each processor. Store these amounts in a array, each entry of the array corresponds to a scanline in the screen.
2. Perform a global vector sum operation on the array in each processor so that the number of pixels to be merged at each scanline is found globally.
3. Perform a prefix sum on the resultant array to form the "prfx_arry[0...N]". The location "i",where i = 1, ... , N in the prfx_arry gives the number of pixels to be merged between scanlines 1 and i, including the scanlines 1 and i. The last location in this array (prfx_arry[N]) holds the total number of pixels to merged.
4. Set prfx_arry[0] = 0, *Start* = 1, *End* = N , D = d.
5. Find a location "i" in the prfx_arry such that

   prfx_arry[i] = (prfx_arry[*End*] − prfx_arry[*Start*-1])/2 + prfx_arry[*Start*-1]

   This location divides the screen at scanline "i" into two partitions such that each partition has almost equal number of pixels.
6. Set *Start* = i+1 for processors in the upper (D−1) dimensional hypercube.
7. Set *End* = i for processors in the lower (D−1) dimensional hypercube.
8. Set D = D−1.
9. Repeat steps 5-8 on the screen partition between scanlines *Start* and *End* until D = 0.

Figure 4: Recursive adaptive subdivision algorithm.

i+1, and if it belongs to lower (D−1) dimensional subcube it sets *End* = i. Hence, processors belonging to different subcubes run the algorithm on different partitions of the screen. This algorithm is well suited to the communication scheme used in the pairwise exchange scheme. Note that, in pairwise exchange scheme, at each exchange step, processors in the lower/upper subcubes send lower/upper parts of the screen and merges pixels in the upper/lower part of the screen. At each iteration step D, for D = *d*,...,1, of the recursive adaptive subdivision scheme, a division point is found for subcubes of dimension D.

This algorithm can be modified to be used in the *pixel merging by all-to-all personalized communication* scheme. Note that, at the end of the execution, each node actually knows which partition of the screen is assigned to itself by the values of *Start* and *End*. Each node, then, performs a global collect operation, which can be performed in $log_2(P)$ concurrent communication steps, on these values to obtain the values in the other nodes. Therefore, at the end of this global collect operation, each node knows the partition of the screen among the nodes. Pixel merging by all-to-all personalized communication scheme can be performed using this partitioning of the screen.

1. Find the amount of winning pixels at each scanline. Store these amounts in a array (ScanPixelCount array), each entry of the array corresponds to a scanline in the screen.
2. Perform a global vector sum operation on the array in each processor so that the number of pixels to be merged at each scanline is found globally.
3. Sort the scanlines with respect to the number of pixels in decreasing order.
4. Set i = 1
5. Assign the scanline i in the sorted array to processor "k" whose current work load is minimum.
6. Update the work load of the processor "k" by incrementing its work load by the number of pixels in scanline "i".
7. Set i = i + 1
8. Repeat steps 5-7 until all scanlines are assigned to processors.
9. Renumber the scanlines such that scanlines assigned to a processor are numbered consecutively.

Figure 5: Algorithm for Heuristic Bin Packing

6.2 Heuristic Bin Packing

In the *pixel merging by all-to-all personalized communication* scheme, the load metric of a node processor in pixel merging operation is the sum of the number of local winning pixels and the winning pixels it receives from other processors after local hidden surface removal step. A partitioning strategy that tries to achieve partition such that this total number is almost equal in each processor, also achieves a good load balancing. In the recursive partitioning scheme, this goal is achieved to some extent. Note that, the division of the screen is done on scanline basis, i.e. scanlines are not divided. For this reason, it is difficult to achieve exactly equal load in each partition. In addition, when a division point is found and screen is divided into two partitions, each partition is subdivided independent from the other one. As a result, at each recursive subdivision, the load imbalance between the partitions may propagate and increase. Therefore, at the end of recursive subdivision, some node processors may still have substantially more work load than others. A more evenly distribution of work load among the nodes can be achieved by using a different partitioning scheme, called heuristic bin packing. In this scheme, the goal is to minimize the difference between the loads of the maximum loaded processor and minimum loaded processor. In order to realize this goal, a scanline is assigned to a processor with minimum work load. In addition, scanlines are assigned in decreasing number of pixels they have, i.e. scanlines that have large number of pixels are assigned at the beginning. In this way, large variations in the processor loads due to new assignments are minimized towards the end. The algorithm, which is executed by each processor, is given in Fig. 5.

At steps 1 and 2 of the algorithm, the total number of pixels at each scanline after local hidden surface removal step is found. Then, scanlines are sorted with respect to number of pixels in decreasing order. This sorting is done in parallel. Assume that the size of the set of scanlines, which have non-zero number of pixels, is S. For parallel sorting, each processor sorts a disjoint subset of size S/P of this set of scanlines in parallel. Then, sorted arrays in each processor are merged to obtain the final sorted array. This merge operation can be performed in $log_2(P)$ concurrent communication steps. In this work, load balancing in parallel sorting operation is not considered. Various parallel sorting algorithms can be found in [10, 11]. Through steps 5 - 8, scanlines are assigned to processors. At step 5, the minimum work loaded processor is found using a binary heap.

In iPSC/2 hypercube multicomputer, if a processor wants to send K bytes of data to another processor, these K bytes should be stored in consecutive memory locations so that data can be transmitted in one send operation. Note that, during local hidden surface removal, the winning pixels are stored into winning pixel array in scanline order in consecutive locations. However, the load balancing algorithm may assign consecutive scanlines to different processors. Hence, non-consecutive scanline data in the winning pixel array of a processor "l" can be assigned to a processor "k". As a result, in order processor "l" to send the pixels belonging to scanlines assigned to processor "k", it has to gather those pixels in another array so that they are stored in consecutive memory locations. In order to avoid this extra gather operation, the algorithm in Fig. 5 is executed before local hidden surface removal. At step 9, scanlines are renumbered so that scanlines assigned to a processor are numbered consecutively. Therefore, pixels generated for these scanlines will be stored in consecutive locations in winning pixel array. However, the load metric in heuristic bin packing algorithm is the number of pixels in each scanline after local hidden surface removal is performed. In order to find the number of winning pixels after local hidden surface removal without running local z-buffer operations, each processor executes the algorithm called *extended span algorithm* given in Fig. 6 on spans in the *span list* structure.

In this algorithm, intersecting spans in scanline "i" are merged to form extended spans. The number of pixels in these extended spans gives the number of winning pixels after local z-buffering for scanline "i". Note that, the number of pixels to be generated for a single span is $(x_r - x_l + 1)$. Remember that during scanline span list creation, spans are sorted with respect to their x_l values in increasing order. Because of the sorting, there is no need to store the extended spans. In addition, checking the intersection of a span "j" with the extended span can be done by only checking x_l of span "j" with extnd_span_x_r.

7 Experimental Results

The algorithms proposed in this work were implemented in C language on a 16-node Intel iPSC/2 hypercube multicomputer. Algorithms were tested for scenes composed of 1, 2, 4, and 8 tea pots for screens of size 400x400, 640x640, and 800x800. Table 1

```
Initialize ScanPixelCount array to zero.
for ( each scanline "i" ) do
   extnd_span_x_r = -1;
   extnd_span_x_l = 0;
   for ( each span "j" in scanline "i" ) do
      if ( x_l < extnd_span_x_r ) then
         if ( x_r > extnd_span_x_r ) then
            extnd_span_x_r = x_r;
         endif
      else
         ScanPixelCount[i] = extnd_span_x_r - extnd_span_x_l + 1;
         extnd_span_x_r = x_r;
         extnd_span_x_l = x_l;
      endif
   endfor
   ScanPixelCount[i] = extnd_span_x_r - extnd_span_x_l + 1;
endfor
```

Figure 6: Extended span algorithm.

Table 1: Scene characteristics in terms of total number of pixels generated (TPG), number of polygons, and total number of winning pixels in the final picture (TWP) for different screen sizes.

Scene	Num. Of Polygons	N=400		N=640		N=800	
		TPG	TWP	TPG	TWP	TPG	TWP
1 POT	3751	59091	43247	137043	110515	206949	172600
2 POT	7502	66802	37084	151881	94840	228170	148191
4 POT_1	15004	71578	26328	146468	66727	211730	103969
4 POT_2	15004	81735	35629	171480	90692	249524	141632
8 POT_1	30008	154187	52258	324464	133617	473417	208731
8 POT_2	30008	99589	36043	201829	91729	289389	143241

gives the characteristics of the scenes in terms of total number of pixels generated, number of polygons and total number of winning pixels in the final picture for different screen sizes. In this table, TPG represents the total number of pixels generated, and TWP represents the total number of winning pixels in the final picture. Other abbreviations in the following tables are, AAPC-HBP: *pixel merging by all-to-all personalized communication scheme using heuristic bin packing for load balancing*, AAPC-RS: *pixel merging by all-to-all personalized communication scheme using recursive adaptive subdivision algorithm for load balancing*, PAIR-RS: *pixel merging by pairwise exchange scheme using recursive adaptive subdivision for load balancing*, ZBUF-EXC: *straightforward implementation which exchanges pixel information for all active and inactive pixel locations.* All timing results in the tables are in milliseconds.

Table 2 illustrates the performance comparison of PAIR-RS scheme straight-forward implementation (ZBUF-EXC) scheme. The timings for some scene instances for

Table 2: Relative execution times of straight-forward implementation and PAIR-RS for N=400.

P	Scene	PAIR-RS			ZBUF-EXC		
		Span List Creation	Local z-buffer	Pixel Merging	Span List Creation	Local z-buffer	Pixel Merging
16	1 POT	322	434	348	316	578	2015
	2 POT	481	471	341	470	585	1940
	4 POT_1	1038	520	323	1015	647	1930
	4 POT_2	1124	579	408	1099	702	1958
	8 POT_1	2142	1079	684	2104	1128	2043
	8 POT_2	2087	701	451	2029	805	1958
8	1 POT	630	815	468	612	952	1941
	2 POT	947	886	475	920	989	1882
	4 POT_1	2037	989	419	1968	1093	1798
	4 POT_2	2268	1109	545	2186	1191	1881
	8 POT_1	4219	2030	861	*	*	*

ZBUF-EXC scheme could not be obtained due to insufficient local memory. Those cases are indicated by a "*" in this table. As is seen in Table 2 table, modified scanline algorithm gives much better results than ZBUF-EXC scheme in pixel merging phase. The pixel merging phase in ZBUF-EXC is similar to that of PAIR-RS scheme. The only difference is that unlike PAIR-RS scheme, pixel information for inactive pixel locations are also exchanged in ZBUF-EXC scheme. Since pixel information for inactive pixel locations are also exchanged, the volume of communication in ZBUF-EXC scheme is larger than that of PAIR-RS. As is also seen from the table, the PAIR-RS scheme performs better than ZBUF-EXC scheme also in local z-buffer phase. Therefore, we may conclude that overheads associated with local z-buffer phase in PAIR-RS scheme is less than that of ZBUF-EXC scheme. In PAIR-RS scheme, the scanline z-buffer is initialized only once. In ZBUF-EXC scheme, it is re-initialized for each scanline on the screen.

Table 3 illustrates the performance comparison of AAPC-HBP, AAPC-RS, and PAIR-RS schemes. The timing results for *local z-buffer* do not include the time spent on span list creation, because all algorithms use the same span list creation algorithm. The overheads associated with load balancing operations are incorporated into *local z-buffer* operation. If we compare the pixel merging times, AAPC-HBP scheme gives the best results among all schemes. This is because of the fact that the *heuristic bin packing* scheme achieves better load balancing than *recursive adaptive subdivision* scheme. As is also seen from the table, PAIR-RS scheme gives worst performance results in pixel merging phase. This is because of the store-and-forward overhead associated with this scheme. If performance of the algorithms are compared with respect to execution time of *local z-buffer* operation, algorithms that use *recursive adaptive subdivision* scheme performs better. This is due to the fact that *recursive adaptive subdivision* scheme introduces less overhead to the execution. However, on overall execution time (local z-buffer time + pixel merging time), the AAPC-HBP scheme achieves almost %6 improvement over AAPC-RS scheme for 8 POT_2 scene for P=16 and N=800.

Table 3: Comparison of execution times of several pixel merging schemes.

N	P	Scene	AAPC-HBP		AAPC-RS		PAIR-RS	
			Local z-buff.	Pixel Merg.	Local z-buff.	Pixel Merg.	Local z-buff.	Pixel Merg.
400	16	4 POT_1	550	181	524	218	520	323
		4 POT_2	608	200	583	247	579	408
		8 POT_1	1126	302	1083	376	1079	684
		8 POT_2	735	212	705	268	701	451
	8	4 POT_1	1031	250	992	291	989	419
		4 POT_2	1150	293	1112	351	1109	545
		8 POT_1	2098	464	2034	543	2030	861
		8 POT_2	1379	301	1334	374	1330	577
640	16	4 POT_1	1060	333	1016	418	1011	702
		4 POT_2	1213	388	1169	488	1164	879
		8 POT_1	2238	611	2170	794	2165	1502
		8 POT_2	1412	404	1363	535	1358	976
	8	4 POT_1	2013	540	1951	636	1947	936
		4 POT_2	2318	627	2257	764	2253	1208
		8 POT_1	4250	1050	4146	1242	4142	1957
		8 POT_2	2686	660	2615	818	2611	1303
800	16	4 POT_1	1500	475	1445	595	1439	1038
		4 POT_2	1708	551	1656	704	1650	1305
		8 POT_1	3179	901	3096	1180	3090	2225
		8 POT_2	1978	551	1918	776	1912	1421
	8	4 POT_1	2842	791	2763	960	2758	1398
		4 POT_2	3305	929	3229	1147	3224	1796

Table 4: Execution times of heuristic bin packing and recursive adaptive subdivision heuristics.

P	N	Scene	Heuristic Bin Packing			Recursive Adaptive Subdivision
			Total Time	Extended Span Alg.	Parallel Sort	
16	400	4 POT_1	60	15	15	20
		4 POT_2	60	19	14	20
		8 POT_1	75	25	17	20
		8 POT_2	64	19	14	21
	640	4 POT_1	87	22	22	28
		4 POT_2	86	23	21	27
		8 POT_1	108	37	23	25
		8 POT_2	90	28	19	27
	800	4 POT_1	101	26	25	30
		4 POT_2	99	27	22	30
		8 POT_1	128	46	27	33
		8 POT_2	107	34	23	30
8	400	4 POT_1	65	24	17	15
		4 POT_2	65	24	15	17
		8 POT_1	91	43	17	15
		8 POT_2	72	30	16	16
	640	4 POT_1	94	35	25	20
		4 POT_2	94	37	23	21
		8 POT_1	135	65	32	20
		8 POT_2	104	47	23	23
	800	4 POT_1	116	43	32	24
		4 POT_2	112	45	27	23

Table 4 illustrates parallel execution times of heuristic bin packing and recursive adaptive subdivision schemes. As is seen from the table, recursive adaptive subdivision scheme introduces less overhead than heuristic bin packing, because overheads of parallel sorting and extended span algorithm in heuristic bin packing are not present in recursive adaptive scheme. As is seen from the table, execution times of extended span algorithm and parallel sorting operation almost take %50 of the total execution time. Note that, the execution time of the recursive adaptive subdivision scheme remains almost constant for a constant screen size, and increases with increasing screen size. This is due to the fact that the execution time of the recursive adaptive subdivision scheme depends on the number of scanlines in the screen. As is seen from the table, as the number of processors increase, the execution time of extended span algorithm decreases as is expected, because its execution time depends on the number of spans in a processor. On the other hand, the execution time of the recursive adaptive subdivision algorithm decreases with decreasing number of processors. For smaller number of processors, screen is divided into smaller number of partitions, hence steps 5-8 are repeated smaller number of times. Also global vector sum and prefix operations take slightly less time for smaller number of processors.

8 Conclusions

In this paper, various algorithms for object-space parallel rendering on hypercube multicomputer are proposed and presented.

A modified scanline z-buffer algorithm is developed to store only the winning pixels in each processor to decrease the number of global z-buffering operations and volume of communication in pixel merging phase. This algorithm also avoids the re-initialization of scanline z-buffer for each scanline on the screen, hence reduces the overhead associated with re-initialization of the scanline z-buffer.

Efficient algorithms are developed for pixel merging phase of the object-space parallel rendering. Two algorithms,which use different interprocessor communication strategies, are developed. The *pixel merging by all-to-all personalized communication* scheme gives better results than *pixel merging by pairwise exchange* scheme due to less store-and-forward overhead. However, number of communication steps of the pixel merging by all-to-all personalized communication scheme, which is $P-1$ steps, is more than that of pixel merging by pairwise exchange scheme, which is $log_2(P)$ steps. Hence, for large number of processors with high set-up time pixel merging by pairwise exchange scheme may be preferable.

In this paper, load balancing issues in pixel merging step is also discussed. Heuristic algorithms are developed to achieve an even distribution of work load among processors. From the experimental results, the *heuristic bin packing* scheme gives better performance results in pixel merging phase, because of better load balancing achieved. In addition, *heuristic bin packing* scheme achieves better performance results also in total execution time (local z-buffer time + pixel merging time) in spite of the more overhead for load balancing operation. Therefore, for object-space parallel rendering on hypercube multicomputers, it is recommended to use *pixel merging by all-to-all personalized communication* scheme along with *heuristic bin packing* load balancing scheme.

References

[1] A. Watt, *Fundamentals of Three-Dimensional Computer Graphics*, Addison Wesley, (1989).

[2] D. F. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, (1985).

[3] S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers, Boston (1992).

[4] M. Kaplan and D. Greenberg, Parallel processing techniques for hidden surface removal. *SIGGRAPH '79*, **13**(2), 300-307 (Aug. 1979).

[5] M. C. Hu and J. D. Foley, Parallel processing approaches to hidden-surface removal in image space. *Comput. & Graphics*, **9**(3), 303-317 (1985).

[6] M. Cox and P. Hanrahan, Pixel merging for object-parallel rendering: A distributed snooping algorithm. In *Proc. of 1993 Parallel Rendering Symposium*, San Jose, 49-56 (1993).

[7] R. Scopigno, A. Paoluzzi, S. Guerrini, and G. Rumolo, Parallel depth-merge: A paradigm for hidden surface removal. *Comput. & Graphics*, **17**(5), 583-592 (1993).

[8] J. Xu and K. Hwang, Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, **18**, 1-13 (1993).

[9] S.F. Nugent, The iPSC/2 direct-connect communication technology. In *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.*, 51-60 (Jan. 1988).

[10] B. Abalı, F. Özgüner, and A. Bataineh, Balanced parallel sort on hypercube multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, **4**(5), 572-581 (1993).

[11] C.G. Plaxton, Load balancing, selection and sorting on the hypercube. In *Proc. of 1989 ACM Symp. Parallel Algorithms and Architectures*, 64-73 (1989).