Pergamon

PII: S0097-8493(98)00047-8

Comput. & Graphics, Vol. 22, No. 4, pp. 487–503, 1998

Printed in Great Britain
0097-8493/98 $19.00 + 0.00

Technical Section

OBJECT-SPACE PARALLEL POLYGON RENDERING ON HYPERCUBES

TAHSIN M. KURÇ[1], CEVDET AYKANAT[2]† and BÜLENT ÖZGÜÇ[2]

[1]Department of Computer Science, University of Maryland, College Park, MD 20742, USA
[2]Department of Computer Engineering and Information Science, Bilkent University, 06533 Ankara, Turkey

Abstract—This paper presents algorithms for object-space parallel polygon rendering on hypercube-connected multicomputers. A modified scanline z-buffer algorithm is proposed for local rendering phase. The proposed algorithm avoids message fragmentation by packing local foremost pixels in consecutive memory locations efficiently, and it eliminates the initialization of scanline z-buffer for each scanline. Several algorithms, utilizing different communication strategies and topological embeddings, are proposed for global z-buffering of local foremost pixels during the pixel merging phase. The performance comparison of these pixel merging algorithms are presented based on the communication overhead incurred in each scheme. Two adaptive screen subdivision heuristics are proposed for load balancing in the pixel merging phase. These heuristics utilize the distribution of foremost pixels on the screen for the subdivision. Experimental results obtained on an Intel's iPSC/2 hypercube multicomputer and a Parsytec CC system are presented. Rendering rates of 300K–700K triangles per second are attained on 16 processors of Parsytec CC system in the rendering of datasets from publicly available SPD database. © 1998 Elsevier Science Ltd. All rights reserved

Key words: polygon rendering, parallel, distributed memory, multicomputers, hypercube.

1. INTRODUCTION

Algorithms and methods in polygon rendering field [1] deal with producing realistic images of computer generated environments composed of polygons. A pipeline of operations is applied to render polygons. These pipeline of operations transform polygons from 3-dimensional (3D) space to 2D screen space, perform smooth shading of the polygons, and perform hidden-surface removal to give realism to the image produced. Among many hidden-surface removal algorithms, *z-buffer* and *scanline z-buffer* algorithms are more popular due to wider range of applications and better utilization of coherency.

Rendering of 3D complex scenes has been a challenge for many years in computer graphics field. Along with the advances in computer graphics, 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 graphical models (e.g., large number of polygons that make up the scene) has required more and more memory. General purpose distributed-memory multicomputers can provide a cost-effective and flexible environment for fast image generation.

Polygon rendering applications can be considered as containing two interacting domains, namely *image-space* and *object-space*. Image-space (screen), on which the result of the rendering is displayed, constitutes the output domain of the rendering process. Object-space is the input dataset defined in 3D space, and it constitutes the input domain of the rendering process. Based on these domains, there are basically two approaches for parallel rendering; *image-space parallelism* and *object-space parallelism*.

In this work, we investigate object-space parallelism for polygon rendering on hypercube-connected multicomputers. In object-space parallelism, the domain of decomposition is the input domain of the rendering process. The primitives (polygons, objects, etc.) that constitute the environment are distributed among the processors. Processors concurrently render their local primitives, thus producing partial images. After *local rendering* phase, partial images in all processors are merged to obtain the final picture because primitives in different processors may contribute to the same pixel location on the screen. *Pixel merging* (*image composition*) phase is performed by exchanging local image buffers fully or partially over the interconnection network. Object-space parallelism is also called *sort-last* approach [2].

In object-space parallelism, efficient parallelization of the pixel merging phase is one of the most critical issues because pixel merging phase introduces overhead to the parallel execution. An architecture with a pipelined image-composition network

†Corresponding author. E-mail: aykanat@cs.bilkent.edu.tr.

to perform pixel merging is presented in [3, 4]. However, full z-buffer in each processor is injected into the communication network resulting in unnecessarily high volumes of communication. The approaches in [5, 6] use tree interconnection topology for the pixel merging phase. The main disadvantage of both approaches is the low processor utilization in pixel merging phase due to the tree topology. Another approach presented in [7] utilizes network broadcast capability for the pixel merging phase. This approach decreases the volume of communication by injecting only the pixel information for 'active' pixel locations in each processor into the network. Furthermore, the volume of communication is also expected to decrease since each processor, which has not yet broadcast its local pixel information, deletes the local hidden pixels. This approach is well suited to architectures with network broadcast capability or with shared memory because the cost of broadcast is small in these machines. However, communication overhead will be high in distributed-memory machines since each active pixel should be broadcast. The second disadvantage is the low processor utilization: once a processor broadcasts its local pixels, it waits idle until the end of pixel merging phase.

Low processor utilization in the pixel merging phase is a common problem in the previous approaches [5–7]. Lee *et al.* [8] address this problem and divide the screen during pixel merging phase on 2D mesh architectures. Static interleaved assignment of scanlines is utilized for load balancing in the pixel merging phase. Adaptive division of the screen for load balancing in pixel merging computations remains as an alternative to be investigated. The communication overhead is another issue which should be considered carefully. Volume of communication can be decreased by exchanging only foremost pixels in each processor. Exchanging foremost pixels rises one important question as how to extract local foremost pixels to avoid message fragmentation in pixel merging phase. No algorithms are presented in the previous works to answer this question. Efficient algorithms to perform extraction of local foremost pixels in the local rendering phase need to be investigated.

In this work, a modified scanline z-buffer algorithm is proposed for local rendering phase. The nice features of the proposed algorithm are as follows. It avoids message fragmentation in pixel merging phase by storing local foremost pixels in consecutive memory locations efficiently. In addition, it eliminates initialization of scanline z-buffer for each scanline, which introduces a sequential overhead to parallel rendering. All of the processors are utilized actively throughout the pixel merging phase by exploiting the interconnection topology of hypercube and by dividing the screen among processors. The volume of communication is decreased by exchanging only local foremost pixels in each processor after local rendering phase. We propose two schemes, called *pairwise exchange* (PAIR) and *all-to-all personalized communication* (AAPC) schemes. PAIR scheme is also referred to as fold or multinode accumulation [9]. PAIR scheme involves a minimum number of communication steps, but it has store-and-forward overhead. AAPC scheme eliminates this overhead by increasing the number of communication steps. Our AAPC scheme differs from 2-phase direct pixel forwarding of Lee *et al.* [8]. Our algorithm is a one-phase algorithm, i.e., pixels are transmitted to destination processors in a single communication phase. Hence, our algorithm avoids the intermediate z-buffering in [8]. We also investigate load balancing in pixel merging phase. Two adaptive screen subdivision heuristics, namely recursive subdivision and heuristic bin packing, are proposed to achieve better load balancing. These heuristics utilize the distribution of foremost pixels on the screen for the subdivision. We present experimental results on an iPSC/2 hypercube multicomputer and a Parsytec CC system. AAPC scheme with heuristic bin packing achieves rendering rates of 300K–700K triangles per second on 16 processors of Parsytec CC system using the scenes from SPD database [10].

Organization of the paper is as follows. Section 2 summarizes the previous work on object-space parallelism. Object-space parallel polygon rendering algorithm is presented in Section 3. Section 4 describes the proposed modified scanline z-buffer algorithm for the local rendering phase. Section 5 presents several algorithms utilizing different communication strategies and topological embeddings for parallel pixel merging on hypercubes. We give a comparison of these schemes based on the communication overhead incurred in each scheme. Section 6 presents two adaptive screen decomposition algorithms for load balancing in the pixel merging phase. Experimental results on an Intel's iPSC/2 hypercube multicomputer are given in Section 7. Results on a Parsytec CC system are presented in Section 8.

2. PREVIOUS WORK ON OBJECT-SPACE PARALLELISM

There are various works both on image-space parallelism [11–16] and object-space parallelism [3–8]. This section summarizes the previous works on object-space parallelism.

Molnar *et al.* [3] and Eyles *et al.* [4] present PixelFlow architecture for object-space parallel rendering. In this architecture, primitives are distributed among a set of identical renderers (*flow units*), which consist of *geometry processor* and *rasterizer* boards. *Image composition network* provides a daisy-chained connection between rasterizer boards of neighboring flow units. During a typical operation, first the screen is divided into smaller regions. Then, geometry processors transform primitives

into screen space and place them into buckets for each screen region. The screen regions are processed one-by-one. For a given screen region, each renderer rasterizes local primitives in the corresponding bucket. After local rasterization, the pixel data is merged over the composition network and loaded into shaders to convert final pixel data into color values. Shaders feed color values to frame buffers for display. Regions of screen are assigned to shaders in a round-robin fashion. Flow units can be designated to operate as shaders, renderers, or frame-buffers by software. PixelFlow architecture is not the only architecture specialized for parallel rendering. There are other architectures such as Pixel-Planes [17] and SGI Onyx2 [18]. All of these architectures use specialized hardware to achieve high rendering rates. The hardware architectures for rendering is out of the scope of this paper. In this paper, we investigate algorithms for general purpose multicomputers, in particular, with hypercube interconnection topology.

Scopigno *et al.* [6] present a parallel hidden-surface removal (HSR) paradigm based on divide-and-conquer approach. The HSR problem is solved by subdividing the problem into equal size subproblems recursively until the size of the subproblem becomes sufficiently small. HSR is done on the subproblem by 'leafHSR processes'. The results of the leafHSR processes are then merged to obtain the final result. Authors present simulation results for tree-based and shared-memory architectures. In tree-based architecture model, each processor is assigned either to a leafHSR process or to a merge process. In shared-memory model, a scheduling processor assigns processors to leafHSR and merge processes.

Li and Miguet [6] present an algorithm for transputers interconnected by a network configured as a tree structure. Pixel merging phase is done using the tree structure. In order to increase processor utilization and reduce memory requirements, the screen is subdivided into horizontal bands and processing of these bands are pipelined. Once a processor finishes the work on a band, it merges the results from its children in the tree and sends the merged band to its parent. Ternary tree, binary tree and unary tree (ring) interconnection topologies are investigated for pixel merging phase.

Cox and Hanrahan [7] propose a pixel merging algorithm developed for architectures with network broadcast capability. In the pixel merging phase, pixel information at each 'active' pixel location, defined as the pixel location covered by at least one local polygon, is broadcast over the network. Starting from processor 1 and continuing in increasing processor numbers, processor k broadcasts the local pixel information in its local active pixel locations to a global frame-buffer and to processors $k+1, k+2,...,P$ that 'snoop' the network to catch pixel information broadcast. Each snooping processor compares the distance values of received pixels with local pixels and eliminates hidden local pixels from further consideration. In this way, the number of pixels broadcast by the next processor is expected to decrease.

In a recent work, Lee *et al.* [8] present several pixel merging algorithms for 2D mesh multicomputers. Their algorithms consist of two stages. In the first stage, the full screen partial images in each processor are divided into r horizontal regions for an $r \times c$ mesh. These regions are concurrently merged along the rings in the rows of the processor mesh to produce the respective subimages. In the second stage, the subimages in each processor are further divided into c horizontal subregions. These subregions are concurrently merged along the rings in the columns to produce the final image. In their first scheme, regions of local full z-buffer are circulated along the rings for merging and forwarding. In the second scheme, the volume of communication is reduced by circulating bounding boxes that cover only active pixels. In their direct pixel forwarding scheme, the partial images are sent directly to the destination processors in two stages. In the local rendering phase, processors store the generated active pixels in the respective send queues according to the screen region assignment for the first stage. That is, no z-buffering is performed during the local rendering phase. In the first stage, these send queues are directly transmitted to their destination processors in the rows by exploiting the cut-through [9] routing capability of the architecture. Then, each processor z-buffers the received pixels by its local active pixels to reduce the volume of communication for the next stage. In the second stage, active pixels in each processor are merged along the columns through direct pixel forwarding as in the first stage. Lee *et al.* [8] also address the load balancing issue in the pixel merging phase. The subregions assigned to processors consist of interleaved scanlines rather than consecutive scanlines for better load balancing.

3. THE PARALLEL ALGORITHM

The following definitions are given for the sake of clarity of the presentation of the parallel algorithm. A pixel location (x,y) on the image plane is said to be *active* if at least one pixel is generated for that location. Otherwise, it is called an *inactive* pixel location. Note that different processors may generate pixels for the same location. A pixel is said to be a *foremost* (winning) pixel, if it is the current pixel whose z value is minimum for the respective active pixel location. At the end of the pixel merging operation there remains *only one* winning pixel for each active pixel location.

The algorithm for object-space parallel polygon rendering consists of the following three phases; *initialization*, *local rendering* and *pixel merging*. In

the initialization phase, polygon information is distributed to node processors by the host processor using scattered assignment scheme. In this scheme, successive polygons in the sequence are assigned to the processors in a round-robin fashion. In the local rendering phase, each processor performs geometry processing, hidden-surface removal and shading for its local polygons. In this work, we propose and use a modified scanline z-buffer algorithm for hidden-surface removal. This algorithm is presented in Section 4.

After local z-buffering, pixels generated in each processor should be merged because multiple processors may produce pixels for the same pixel location. The global z-buffering operations during the pixel merging phase can be considered as an overhead to the sequential rendering. Each global z-buffering operation necessitates interprocessor communication. Efficient implementation of the pixel merging phase is thus a crucial factor for the performance of object-space parallel rendering. In its simplest form, pixel merging phase can be performed by exchanging pixel information for all pixel locations between processors. We call this scheme *full z-buffer merging*. This scheme may introduce large communication overhead in pixel merging phase because pixel information for inactive pixel locations are also exchanged. This overhead can be reduced by exchanging only local foremost pixels in each processor. This scheme is referred to here as *active pixel merging*.

The motivation behind local z-buffering is to reduce the volume of communication during pixel merging phase through decreasing the number of local pixels to be globally z-buffered. Thus, the benefit of local foremost pixel concept is expected to increase with increasing depth complexity of the scene. However, it should be noted here that local foremost pixel concept does not integrate transparency. Depth-sorted non-opaque pixels obtained during local z-buffering cannot be blended locally because of the possibility of multiple processors generating pixels for the same location. In this case, pixel merging involves merge sorting of the locally sorted pixel lists. Hence, local z-buffering cannot reduce the volume of communication in the merging of non-opaque pixels. However, local z-buffering together with the concept of local foremost opaque pixel can be beneficial in the parallel rendering of hybrid scenes containing both opaque and non-opaque primitives. During local z-buffering, pure z-buffering is adopted between opaque pixels to maintain the current foremost opaque pixel, whereas depth sorting is adopted for non-opaque pixels which are not obstructed by the current foremost opaque pixel. This local z-buffering scheme can easily be implemented by maintaining a linked list of depth-sorted pixels for each local active pixel location such that each linked list will contain at most one opaque pixel as its last entry. In this way, all local opaque and non-opaque pixels obstructed by the local foremost opaque pixels will be avoided from global merging, thus reducing the volume of communication. The algorithms presented in the rest of the paper assume that the scene is composed of only opaque primitives.

4. A MODIFIED SCANLINE z-BUFFER ALGORITHM

In distributed-memory multicomputers, transmitting all data elements in one send operation takes less time than transmitting each element in distinct steps due to setup time of each message. In order to prevent message fragmentation in active pixel merging, the local foremost pixels should be stored in consecutive memory locations. In this section, we propose and present a *modified scanline z-buffer* algorithm which stores foremost pixels in consecutive memory locations efficiently. The proposed algorithm also avoids initialization of the scanline z-buffer for each scanline.

When polygons are projected onto the screen, some of the scanlines intersect the edges of the projected polygons. Each pair of such intersections is called a *span*. In the first phase of the proposed algorithm, these *spans* are generated and inserted into the local *scanline span lists* (SSL) structure. SSL is a 1D virtual array that holds a linked list of local polygon spans for each scanline. Each span is represented by a record, which contains the intersection pair (minimum x-intersection x_{min} and maximum x-intersection x_{max}) and necessary information for z-buffering and shading through span rasterization. SSL is constructed by inserting the spans of the projected polygons to the appropriate scanline lists in sorted (increasing) order according to their x_{min} values. This sorting allows to perform local z-buffering without initializing the scanline array for each scanline on the screen.

In the second phase, spans in the SSL structure are processed, in scanline order (y order), for local z-buffering and shading. Two local 1D arrays are used to store only local foremost pixels. These two local arrays are called *Winning Pixel Array* (WPA) and *Modified Scanline Array* (MSA). WPA stores the information about the foremost (winning) pixels. Each entry in this array contains location information, z value and shading information about the respective local foremost pixel. Since z-buffering is done in scanline order, the pixels in 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. MSA is a modified scanline z-buffer. It is an integer array of size N for a screen of resolution $N \times N$. MSA[x] gives the index of the pixel generated at location x in WPA. At the beginning, each entry of the MSA is set to zero. Moreover, a *range* value is associated with each scanline. The range value of the current

scanline is set to one plus the index of the last pixel, which is generated by the previous scanline, in WPA. The range value for the first scanline is set to 1. Since spans are sorted in increasing x_{min} values, if a location x in MSA has a value less than the range value of the current scanline, it means that location x is generated by a span belonging to previous scanlines. For such locations, the generated pixels are directly stored into WPA without any comparison. Otherwise, the generated pixel is compared with the pixel pointed by the index value. This indexing scheme and sorted order of spans in the SSL structure avoid re-initialization of MSA at each scanline. However, due to comparison made with the range value, an extra comparison is introduced for each pixel generated. These extra comparison operations are reduced as follows. The sorted order of spans in the SSL structure assures that when a span s in scanline y is rasterized, it will not generate a pixel location x which is less than x_{min} of the previous spans. The current span s is divided into two segments such that one of the segments cover the pixels generated by the previous spans in the current scanline and other segment covers the pixels generated by the spans in the previous scanline. Distance comparisons are made only for the pixels in the first segment. The pixels generated for the second segment are stored into WPA without any distance comparisons.

5. PIXEL MERGING ON HYPERCUBES

This section presents pixel merging algorithms developed for a d-dimensional hypercube multicomputer with $P = 2^d$ processors. In these algorithms, each processor initially owns local foremost pixels belonging to the whole screen of size $N \times N$. Then, a global z-buffering operation is performed so that each processor gathers pixels belonging to a horizontal screen subregion of size $N \times (N/P)$.

The algorithms presented in this section use different interprocessor communication strategies and different interconnection topologies that can be embedded onto hypercube. The communication overhead of each algorithm is analyzed for *full z-buffer merging* and *active pixel merging* schemes. For full z-buffer merging, it is assumed that there are $A = N \times N$ pixel locations on the screen. For active pixel merging, we assume that each processor has F foremost pixels after local z-buffering, which are distributed evenly on the image-space along y-dimension, and we also assume that processors are perfectly load balanced at each communication step. Perfect load balance and even distribution assumptions are made to simplify the analysis of each algorithm.

In the equations given in the following sections, t_{su} denotes the setup time for a message, t_{trfull} denotes the time to transmit one pixel location on z-buffer, and $t_{tractive}$ denotes the time required to transmit one active pixel information. A pixel location on z-buffer contains depth value (z) and color values red, green, and blue. An active pixel information contains x position of the pixel in addition to z and color values.

5.1. *Ring exchange scheme*

One way of performing pixel merging is to embed a ring onto the hypercube using gray-code ordering [19], and perform the pixel merging on the ring. In the ring exchange scheme, each processor receives pixels from its right neighbor and sends pixels to its left neighbor. In this scheme, the screen is divided into P regions and numbered from 0 to $P-1$. At exchange step i ($i = 1,...,P-1$), kth processor in the ring transmits the pixels in the region ($k + i$) *mod* P to its left neighbor and receives the pixels in the region ($k + i + 1$) *mod* P from its right neighbor. The receiving processor merges the pixels in the received screen region with the local region and stores them in order to transmit in the next step. These exchange operations are repeated $P-1$ times.

In full z-buffer merging, A/P pixels are concurrently sent and received at each communication step. The communication time in this scheme is

$$T_{comm} = (P-1)t_{su} + \frac{P-1}{P} A t_{trfull}. \quad (1)$$

In *active pixel merging*, each processor sends only the foremost pixels to its left neighbor and receives only the active pixels from its right neighbor. The receiving processor merges these pixels with the local foremost pixels. The number of pixels after this merge operation is equal to the number of active pixel locations in the union of two sets: set of local active pixel locations and set of received pixel locations in the respective screen region. If the processor has L foremost pixels for a screen region and receives R pixels for the same region, then at the end of the merge operation at step i, the number of foremost pixels will be $L + C_i$, where $0 \le C_i \le R$, assuming $R \le L$. If two sets are totally disjoint then no pixels are merged, making C_i equal to R. In other words, C_i represents the amount of concurrent store-and-forward overhead due to the pixels that do not merge at the ith concurrent store-merge-and-forward step. Therefore, the communication time in active pixel merging is

$$T_{comm} = (P-1)t_{su} + \left(\frac{P-1}{P} F + \sum_{i=1}^{P-2} (P-i-1)C_i\right) t_{tractive}. \quad (2)$$

As seen in Equation (2), the volume of communication in active pixel merging depends both on the number of local foremost pixels and the distribution of pixels in the subregion for which merging is performed.

5.2. *2-dimensional mesh exchange scheme*

A 2-D mesh with $M = 2^{\lceil d/2 \rceil}$ columns and $K = 2^{\lfloor d/2 \rfloor}$ rows can be embedded onto a hypercube with $P = M \times K$ processors [19]. In mesh embedding, each row and each column of the mesh form rings in gray-code ordering. Pixel merging can be done using these rings in the mesh embedding. First, the screen is divided into M regions. The processors at each row, independently from other rows, merge these M regions along the respective row. After rowwise merging, nodes on the same column have the same screen region of size A/M pixels. Each of these screen regions are further divided into K regions, and pixel merging is done along the columns of the mesh.

The communication time (T_{comm}) required for a 2D mesh exchange scheme is the sum of the communication time required for rowwise merging (T_{row}) and columnwise merging (T_{column}). Since rows and columns are simply rings, we can use the equations for ring exchange scheme. In full z-buffer merging, A/M pixels are concurrently sent and received at each exchange step of the rowwise merging stage. Hence, the communication time for rowwise exchanges is

$$T_{row} = (M-1)t_{su} + \frac{M-1}{M} A t_{trfull}. \tag{3}$$

After rowwise merging, each screen region is further divided into K subregions. Hence, in full z-buffer merge, $A/(MK)$ pixels are concurrently transmitted and received at each exchange step of columnwise merging. As a result, the communication time for columnwise exchanges is

$$T_{column} = (K-1)t_{su} + \frac{K-1}{MK} A t_{trfull}. \tag{4}$$

Hence, total communication time in full z-buffer merging is

$$\begin{aligned} T_{comm} &= T_{row} + T_{column} \\ &= (M+K-2)t_{su} + \frac{P-1}{P} A t_{trfull}. \end{aligned} \tag{5}$$

Using a similar approach, communication time for rowwise exchanges in active pixel merging is

$$T_{row} = (M-1)t_{su} + \left(\frac{M-1}{M} F + \sum_{i=1}^{M-2} (M-i-1)C_i \right) t_{tractive}. \tag{6}$$

After rowwise merging, the remaining number of foremost pixels ($L_{foremost}$) at each processor is

$$L_{foremost} = \frac{F}{M} + \sum_{i=1}^{M-1} C_i. \tag{7}$$

As in full z-buffer merging, the remaining pixel set is further divided to exchange along the columns of the mesh. Therefore, the communication time for columnwise merging is

$$\begin{aligned} T_{column} =& (K-1)t_{su} \\ &+ \left(\frac{K-1}{K} L_{foremost} + \sum_{i=1}^{K-2} (K-i-1)B_i \right) t_{tractive} \\ &(K-1)t_{su} + \left(\frac{K-1}{P} F + \frac{K-1}{K} \sum_{i=1}^{M-1} C_i \right. \\ &\left. + \sum_{i=1}^{K-2} (K-i-1)B_i \right) t_{tractive}. \end{aligned} \tag{8}$$

Here, C_i and B_i denote the amount of pixel store-and-forward overhead at step i of the rowwise and columnwise merging phases, respectively. As a result, total communication time in active pixel merging is

$$\begin{aligned} T_{comm} =& (M+K-2)t_{su} \\ &+ \left(\frac{P-1}{P} F + \sum_{i=1}^{M-2} (M-i-1)C_i \right. \\ &\left. + \frac{K-1}{K} \sum_{i=1}^{M-1} C_i + \sum_{i=1}^{K-2} (K-i-1)B_i \right) t_{tractive}. \end{aligned} \tag{9}$$

The 2D mesh scheme is a generalized version of ring exchange scheme since a ring can be considered as a 2D mesh with $M = P$ and $K = 1$. It is possible to embed meshes of higher dimensions onto the hypercube [19]. In the following section, a general k-dimensional mesh exchange scheme is derived and analyzed.

5.3. *k-Dimensional mesh exchange scheme*

Assume we embed a k-dimensional mesh onto the hypercube as $P = 2^d = \prod_{i=0}^{d-1} L_i$. Here, L_i represents the number of processors in the ith dimension of the mesh with $L_i \neq 1$ for $i = 0,...,k-1$ and

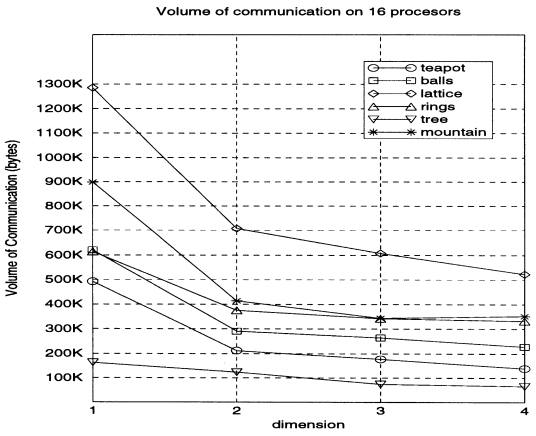


Fig. 1. Concurrent communication volume (in bytes) on different meshes embedded onto a 4-dimensional hypercube with 16 processors for different scenes

$L_i = 1$ for $i = k,...,d-1$. A ring is obtained by making $L_0 = P$ and $L_i = 1$ for $i = 1,...,d-1$. In the k-dimensional mesh, a similar exchange scheme as in 2D mesh exchange is applied. That is, pixel merging is done along the rings embedded at each dimension. At stage i of the pixel merging in k-dimensional mesh, the rings embedded in dimension i is utilized to perform the pixel merging.

In full z-buffer merging, communication time is equal to the sum of communication times at each stage. The communication time (T_i) at stage i is equal to the communication time for pixel merging along the corresponding ring in dimension i of the k-dimensional mesh:

$$T_i = (L_i - 1)t_{su} + \frac{L_i - 1}{\prod_{j=0}^{i} L_j} A t_{trfull}. \tag{10}$$

Thus, total communication time in full z-buffer merging is

$$T_{comm} = \sum_{i=0}^{k-1} T_i = \sum_{i=0}^{k-1} (L_i - 1)t_{su} + \frac{P-1}{P} A t_{trfull}. \tag{11}$$

In active pixel merging, the communication time at stage i is

$$T_i = (L_i - 1)t_{su} + V_i t_{tractive}, \tag{12}$$

where concurrent communication volume (V_i) is

$$V_i = \frac{L_i - 1}{\prod_{j=0}^{i} L_i} F + \sum_{j=0}^{i-1} \left(\frac{L_i - 1}{\prod_{\ell=j+1}^{i} L_\ell} \sum_{n=1}^{L_j - 1} C^{j_n} \right) + \sum_{j=1}^{L_i - 2} C_j^i (L_i - j - 1). \tag{13}$$

Here, C_j^i represents volumes of communication incurred due to the distribution of active pixel locations in a region at the communication step j along the ring embedded in dimension i of the mesh.

The first and second terms in Equation (13) represent the volume of communication incurred due to the active pixel locations in each processor before stage i. The last term in the equation represents the volume of communication incurred due to the distribution of active pixels in a region in each processor. This term also affects the volume of communication in the later stages of the pixel merging since it affects the number of active pixels in a processor after stage i. Therefore, if the volume of communication due to this term is minimized at each stage, the total volume of communication is expected to decrease. One way to minimize the value of this term is to control the distribution of active pixel locations in each region. Controlling the active pixel distribution requires a preprocessing step before the distribution of primitives to processors. This preprocessing results in redistribution of polygons between processors before local z-buffering. Note that this preprocessing step should be repeated when viewing direction and orientation change. Another way to minimize the value of the last term in Equation (13) is to minimize the value of L_i at each stage. The last term is minimized when $L_i = 2$ (for $i = 0,...,d-1$) is chosen for the rings at each dimension and a d-dimensional mesh is embedded onto the hypercube.

Figure 1 illustrates volume of communication on different k-dimensional meshes embedded onto a 4-dimensional hypercube for different scenes (see Fig. 9 for the rendered images of the scenes). As seen in Fig. 1, communication volume decreases with increasing mesh dimension. The lowest communication volume is achieved on 4D mesh while the highest is obtained on 1D mesh, i.e., ring exchange scheme. This figure supports our discussion and analysis in this section that the lowest communication volume is expected to occur when a d-dimensional mesh is embedded onto a d-dimensional hypercube. The scheme to implement pixel merging on the d-dimensional mesh (with $L_i = 2$) on hypercube is given in the next section. This scheme is called *pairwise exchange scheme*.

5.4. *Pairwise exchange scheme*

Pairwise exchange (PAIR) scheme exploits the *recursive-halving* idea widely used in hypercube-specific global operations. This operation requires d concurrent divide-and-exchange stages. At each stage i (for $i = 0,1,2,...,d-1$), each processor divides horizontally its current active region of size $N \times n$ into two equal sized subregions (each of size $N \times n/2$), referred to here as top and bottom subregions, where $n = N$ during the initial halving stage. Meanwhile, each processor divides its current local foremost pixels into two subsets as belonging to these two subregions, which are referred here as top and bottom pixel subsets. Then, processor pairs which are neighbors across channel i exchange their top and bottom pixel subsets. After the exchange, processors concurrently perform z-buffering operations between retained and received pixel subsets to finish the stage.

In *full z-buffer merging*, half of the current screen is transmitted and merged at each exchange stage. Therefore, the total time required for interprocessor communication is

$$T_{comm} = dt_{su} + \sum_{d-1}^{i=0} \frac{A}{2^{(i+1)}} t_{trfull} = dt_{su} + \frac{P-1}{P} A t_{trfull}. \tag{14}$$

In *active pixel merging*, each processor transmits half of its current foremost pixels at each exchange

stage. Assuming perfect load balance at each exchange step, the communication time in active pixel merging is

$$T_{comm} = dt_{su} + \left(\frac{P-1}{P} F + \sum_{i=0}^{d-2} \frac{2^{(d-i-1)} - 1}{2^{(d-i-1)}} C_1^{i+1} \right) t_{tractive}. \quad (15)$$

5.5. *All-to-all personalized communication scheme*

All of the schemes discussed so far are *store-merge-and-forward* schemes. 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 foremost 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. During this store-merge-and-forward steps, some pixels may be copied from memory of one processor to memory of the other processors without any merging more than once as shown by the B_i and C_i terms in the equations. This memory-to-memory copy overhead due to the store-and-forward operations can be avoided by sending the pixels directly to their destination processors. This section presents a scheme called *all-to-all personalized communication* (AAPC) to implement this direct pixel forwarding idea.

The iPSC/2 hypercube multicomputer has the cut-through [9] routing capability. So, multi-hop communication between two non-neighbor processors is almost as fast as single-hop neighbor communications if all the links between two processors are not currently used by other messages. The communication hardware uses the e-cube routing algorithm [20]. In an AAPC scheme, the screen is divided into P regions and kth region is assigned to processor k for $k = 0,1,...,P-1$. Each processor simply performs $P-1$ communication steps exchanging pixel data according to the region assignment with a different processor at every step. Each processor must choose its communication partner at each step so that the hypercube links do not suffer congestion. A congestion-free schedule for AAPC using e-cube routing is given in [9, 20]. In this schedule, processor k sends its local pixel data belonging to the screen region $k \oplus i$ directly to processor $k \oplus i$ at exchange step i (for $i = 1,...,P-1$), where '$\oplus$' denotes the bitwise exclusive-or operation. After $P-1$ exchange steps, each processor gathers all of the foremost pixels belonging to its assigned screen region. Then, each processor z-buffers the local pixels and the pixels it receives from other processors through maintaining a local z-buffer of size $N \times (N/P)$. Local pixels are scattered onto the z-buffer without any distance comparisons. The z value of each received pixel is compared with the z value in the respective pixel location in the z-buffer. After all the pixels are processed local z-buffers contain the winning pixels for the final images corresponding to respective screen regions.

In full z-buffer merging, A/P pixels are concurrently exchanged at each communication step. Thus, the communication time in this scheme is

$$T_{comm} = (P-1)t_{su} + \frac{P-1}{P} A t_{trfull}. \quad (16)$$

In active pixel merging, F/P pixels are concurrently exchanged at each communication step. Hence, the communication time in this scheme is

$$T_{comm} = (P-1)t_{su} + \frac{P-1}{P} F t_{tractive}. \quad (17)$$

5.6. *Comparison of pixel merging schemes*

As seen in Equations (1), (5), (11), (14) and (16), the volume of communication in full z-buffer merging is not affected by the distribution of foremost pixels in screen regions. All schemes induce the same concurrent communication volume of $A(P-1)/P$ in full z-buffer merging. However, PAIR scheme induces the smallest number of concurrent communications ($\log_2 P$ as shown in Equation (14)). Hence, PAIR is the most suitable scheme for full z-buffer merging on hypercubes.

As seen in Equations (2), (9), (13) and (15), the volume of communication in active pixel merging is affected by the distribution of pixels in all of the *store-merge-and-forward* schemes, PAIR scheme (Equation (15)) being the least affected one. On the other hand, the volume of communication in an AAPC scheme is not affected by the distribution of pixels as seen in Equation (17). Hence, among all schemes, the AAPC scheme is expected to give the lowest concurrent communication volume in active pixel merging. For large numbers of processors with high communication latency, the number of communication steps, which directly affects the total setup time, is also a crucial factor in the performance of pixel merging. The number of concurrent communication steps is equal to $\log_2 P$ in a PAIR scheme, whereas it is equal to $P-1$ in an AAPC scheme. For large number of processors, the number of communication steps may be a dominating factor in the communication time in the active pixel merging phase. Therefore, among all schemes presented in this section, PAIR and AAPC schemes are most suitable for pixel merging on hypercube multicomputers. Only these two schemes are experimentally investigated in this work.

6. LOAD BALANCING IN ACTIVE PIXEL MERGING

In this section, two heuristics that implement adaptive subdivision of screen among processors to achieve good load balance in active pixel merging are presented.

6.1. *Recursive adaptive subdivision*

Recursive adaptive subdivision (RS) scheme recursively divides the screen into two subregions such that the number of pixels in one subregion is equal to the number of pixels in the other subregion as much as possible. This scheme is well suited to the recursive structure of the hypercube and can be done in parallel. Each processor counts the number of local foremost pixels at each scanline and stores them into a local workload array of size N. Each entry of the array stores the number of local foremost pixels at the corresponding scanline. An element-by-element global sum operation is performed on these local arrays to obtain the distribution of foremost pixels in all processors. Then, using this global workload array, each processor divides the screen into two horizontal bands of consecutive scanlines so that each region contains almost equal number of active pixel locations. Along with the division of the screen, the hypercube is also divided into two equal subcubes of dimension $d-1$. Top subregion is assigned to one subcube while the bottom subregion is assigned to the other subcube. Subcubes perform subdivision of the local subregions concurrently and independently. Since the screen is divided into horizontal bands, the global workload array is re-used during the further subdivision steps.

6.2. *Heuristic bin packing*

In an RS scheme, the subdivision of the screen is done on a scanline basis and consecutive scanlines are assigned to processors. For this reason, perfect load balance cannot be obtained during the recursive bisection steps. As the recursive bisection steps proceed independently, the load imbalance incurred in a particular bisection may propagate and accumulate during the further bisections of the respective pair of subregions. A better distribution of workload among processors can be achieved by using a direct P-way subdivision scheme which allows non-consecutive scanline assignment to processors. A *heuristic bin packing* (HBP) approach is used to minimize the load of the most heavily loaded processor in the subdivision. In order to realize this goal, a scanline is assigned to a processor with minimum workload. In addition, scanlines are assigned in the order of 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.

In each processor, the total number of pixels at each scanline after local hidden surface removal step is found. Then, scanlines are sorted with respect to their pixel counts 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 d concurrent communication steps. In this work, load balancing in a parallel sorting operation is not considered. Various parallel sorting algorithms can be found in [21, 22]. In our HBP implementation, a binary heap is used to find the processor with minimum workload during the scanline assignment process.

As mentioned earlier, in our modified scanline z-buffer algorithm, each processor stores its local foremost pixels into its local winning pixel array (WPA) in scanline order in consecutive locations. However, the HBP algorithm may assign consecutive scanlines to different processors for a better load balance. Hence, non-consecutive scanline data in the local WPA of a processor k can be assigned to another processor ℓ. As a result, in order for processor k to send the pixels belonging to scanlines assigned to processor ℓ, 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 overhead before each send operation, the load balancing algorithm HBP is executed before local hidden surface removal. Then, scanlines are renumbered so that scanlines assigned to every processor are numbered consecutively. In this way, pixels generated for these scanlines are stored in consecutive locations in the local WPAs. However, the load metric in the HBP algorithm is the number of active 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 *extended span* algorithm given in Fig. 2 on the spans in its local *scanline span list* (SSL) structure.

In Fig. 2, subscripts 'ℓ' and 'r' denote the left and right end-points, respectively, of a span (s) and an extended-span (es) in terms of the pixel location in a scanline. In this algorithm, intersecting spans in scanline y are merged to form extended spans. The sum of the number of pixels in these extended spans gives the number of winning pixels $W[y]$ after local z-buffering for scanline y. Recall that during SSL creation, spans are sorted with respect to their x_ℓ (x_{min}) values in increasing order. Because of this sorted order of spans in local SSLs, there is no need to store the extended spans, and checking the intersection of a span s with the current extended-span es can easily be done by only checking x_ℓ of span s with es_r.

7. EXPERIMENTAL RESULTS ON AN iPSC/2 HYPERCUBE

The algorithms proposed in this work were implemented on a 4-dimensional Intel iPSC/2 hypercube multicomputer. Our iPSC/2 system contains 16 nodes each of which is equipped with an 80386/

```
Initialize scanline pixel count array W to zero.
for each scanline y do
    es_ℓ ← 0;
    es_r ← −1;
    for each span s in SSL[y] in sorted order do
        x_ℓ ← x_min of s;
        x_r ← x_max of s;
        if x_l < es_r then
            if x_r > es_r then
                es_r ← x_r;
        else
            W[y] ← W[y] + (es_r − es_ℓ + 1);
            es_ℓ ← x_ℓ;
            es_r ← x_r;
    W[y] ← W[y] + (es_r − es_ℓ + 1);
```

Fig. 2. Extended span algorithm for computing the active pixel counts of scanlines before running local z-buffering

387 processor and 4 MB memory. The hypercube interconnection network implements cut-through routing through direct-connect communication technology [20]. The algorithms were implemented using C language and the native message passing library (NX) of iPSC/2. The algorithms were tested in parallel rendering of scenes composed of 1, 2, 4, and 8 tea pots for screens of size 400×400 and 640×640. Table 1 gives the characteristics of the scenes in terms of total number of polygons, total number of pixels generated and total number of winning pixels in the final picture for different screen sizes. Rendered images of the scenes from the viewing directions used in the experiments are given in Fig. 8.

Here, we mainly present and discuss the experimental performance comparison of the active pixel merging schemes proposed in this work. Full z-buffer merging is also implemented, and only its speedup performance is compared to that of the active pixel merging for the sake of experimental validation of the theoretical analysis given in Section 5. Pairwise exchange scheme is used in the implementation of full z-buffer merging since it is found to be the most suitable scheme for full z-buffer merging on hypercubes (see Section 5.6). The abbreviations used in the figures and tables are AAPC: *all-to-all personalized communication*, PAIR: *pairwise exchange*, RS: *recursive subdivision*, HBP: *heuristic bin packing*, and ZBUF-EXC: *full z-buffer exchange*. All timing results in the tables are in milliseconds.

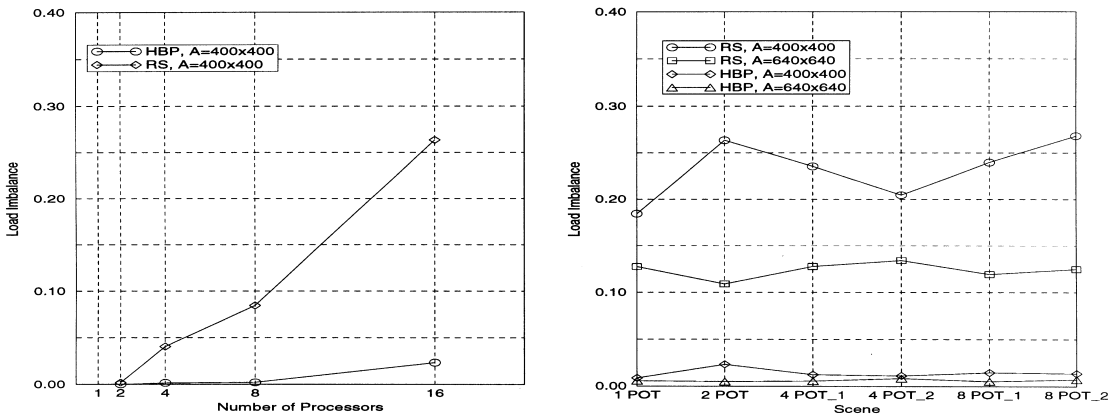
Table 2 illustrates the performance comparison of the active pixel merging schemes AAPC-HBP, AAPC-RS and PAIR-RS. The timing results for local z-buffering step do not include the time spent for SSL creation, because all algorithms use the same span-list creation algorithm. The overheads associated with load balancing operations are incorporated into the local z-buffering time. If we compare the pixel merging times, the AAPC-HBP scheme gives the best results among all schemes. This is because of the fact that the HBP scheme achieves better load balancing than a RS scheme. As also seen in the table, PAIR-RS scheme gives the worst performance results in pixel merging phase. This is because of the store-and-forward

Table 1. Scene characteristics in terms of number of triangles, total number of pixels generated (TPG), and total number of winning pixels in the final picture (TPF) for different screen sizes of $N \times N$

Scene	Number of triangles	$N = 400$		$N = 640$	
		TPG	TPF	TPG	TPF
1 POT	3751	59091	43247	137043	110515
2 POT	7502	66802	37084	151881	94840
4 POT_1	15004	71578	26328	146468	66727
4 POT_2	15004	81735	35629	171480	90692
8 POT_1	30008	154187	52258	324464	133617
8 POT_2	30008	99589	36043	201829	91729

Table 2. Comparison of execution times (in milliseconds) of several active pixel merging schemes

			AAPC-HBP			AAPC-RS			PAIR-RS		
N	P	Scene	Local z-buff.	Pixel merg.	Total	Local z-buff.	Pixel merg.	Total	Local z-buff.	Pixel merg.	Total
	16	4 POT_1	550	181	731	524	218	742	520	323	843
		8 POT_1	1126	302	1428	1083	376	1459	1079	684	1763
400	8	4 POT_1	1031	250	1281	992	291	1283	989	419	1408
		8 POT_1	2098	464	2562	2034	543	2577	2030	861	2891
	16	4 POT_1	1060	333	1393	1016	418	1434	1011	702	1713
		8 POT_1	2238	611	2849	2170	794	2964	2165	1502	3667
640	8	4 POT_1	2013	540	2553	1951	636	2587	1947	936	2883
		8 POT_1	4250	1050	5300	4146	1242	5388	4142	1957	6099



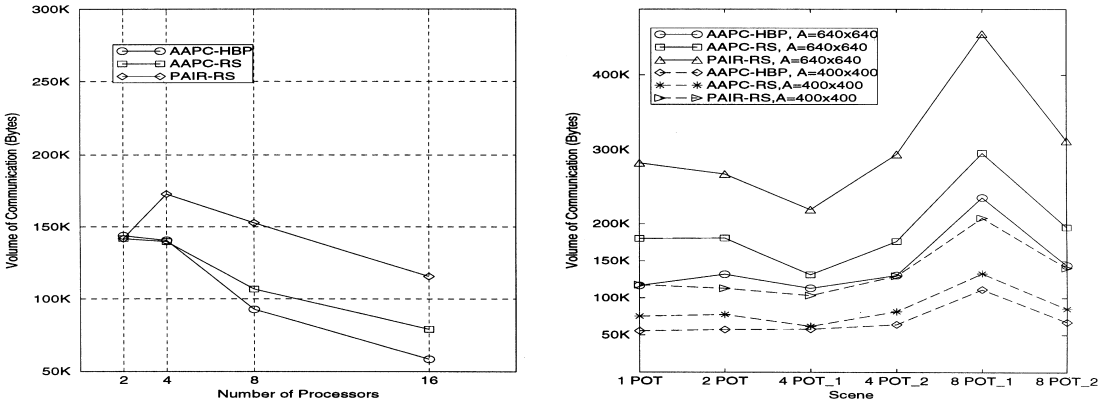
(a) (b)

Fig. 3. Load balancing performance of RS and HBP in active pixel merging for: (a) 2 POT scene on different number of processors at $A = 400 \times 400$, and (b) different scenes at different screen resolutions on $P = 16$ processors

overhead associated with this scheme. If the performance of the algorithms are compared with respect to their local z-buffering time, algorithms that use RS schemes perform better. This is due to the fact that RS schemes introduces less subdivision overhead than HBP schemes. In total (local z-buffering + pixel merging) execution time (*Total*), a AAPC-HBP scheme achieves the best performance in all instances.

Figure 3 illustrates the performance comparison of load balancing heuristics RS and HBP in active pixel merging schemes. The *load imbalance* value is computed as the ratio of the difference between the loads of maximum and minimum loaded processors to average workload. The workload of a processor is taken to be the number of pixel merging operations it performs in the pixel merging phase. As seen in Fig. 3, HBP achieves much better load balance than RS, and as seen in Fig. 3(a) the performance gap between these two schemes rapidly increases with increasing number of processors (P) in favor of HBP. In other words, HBP scales much better than RS as expected since the amount of load imbalance propagation and accumulation rapidly increases with increasing P in RS. As seen in Fig. 3(b), load balancing performance of both RS and HBP schemes improve with increasing screen resolution due to larger flexibility in screen subdivision.

Figure 4 illustrates the total concurrent communication volume (in bytes) for various active pixel merging schemes. The total volume of concurrent communication is calculated as the sum of the maximum volume of concurrent communication at each communication step. As seen in the figure, an AAPC scheme results in substantially less volume



(a) (b)

Fig. 4. Concurrent communication volume (in bytes) for: (a) 2 POT scene on different number of processors at $A = 400 \times 400$, and (b) different scenes on $P = 16$ processors at $A = 400 \times 400$ and $A = 640 \times 640$

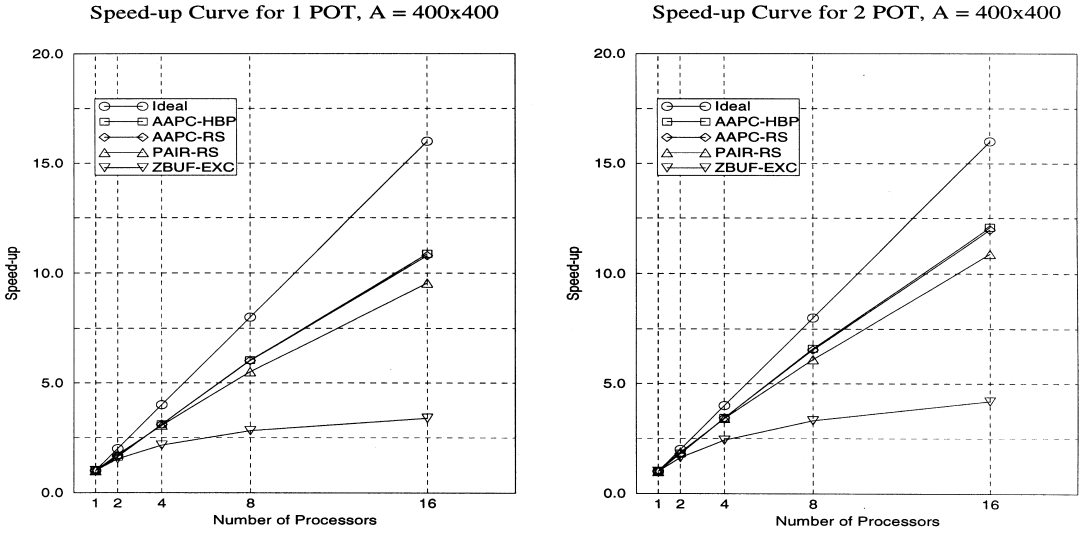


Fig. 5. Speedup figures for (a) 1 POT scene and (b) 2 POT scene at $A = 400 \times 400$

of communication than PAIR scheme as expected. Note that communication volume in active pixel merging is proportional to the number of active pixel locations in each processor. As the number of processors increases, the number of active pixel locations per processor is expected to decrease. Hence, concurrent communication volume is expected to decrease with increasing number of processors as is also seen in Fig. 4(a). The increase in the communication volume of the PAIR-RS scheme as the number of processors increases from 2 to 4 is due to the increase in the store-and-forward overhead. It is also experimentally observed that better load balance in pixel merging leads to less concurrent volume of communication. As seen in Fig. 4(b), a HBP scheme, which achieves better load balancing than RS, results in less volume of communication than an RS scheme in all rendering instances. This is because of the fact that balancing the computational loads of the processors also balances the communication loads of the processors thus reducing the concurrent communication volume.

Figure 5 illustrates speedup curves for different pixel merging schemes. Due to insufficient local memory in node processors of iPSC/2, speedup figures for ZBUF-EXC scheme can only be obtained for 1 POT and 2 POT scenes at screen of size 400×400. Hence, speedup curves for only these two rendering instances are illustrated in the figure for the sake of performance comparison on a common framework. Figure 5 represents the speedup curves for total execution times (span list creation + local z-buffering + pixel merging). As seen in the figure, all active pixel merging schemes achieve substantially better speedup than the full z-buffer merging scheme ZBUF-EXC thus confirming the theoretical results given in Section 5. Since pixel information for inactive pixel locations are also exchanged in full z-buffer scheme, this scheme incurs substantially larger volumes of communication than active pixel merging schemes. As expected, the performance gap between full z-buffer and active pixel merging schemes increases with the increasing number of processors (P) in favor of the active pixel merging schemes. In other words, active pixel merging schemes scale much better than full z-buffer schemes. Concurrent communication volume is expected to decrease with increasing P in active pixel merging, whereas it slowly increases towards the screen size A with increasing P in full pixel merging (see Equation (14)).

As seen in Fig. 5, AAPC schemes achieve considerably better speedup than PAIR schemes in active pixel merging. This is because of the fact that AAPC incurs less volume of communication and smaller numbers of global z-buffering operations than PAIR by avoiding the store-and-forward overhead. Among AAPC schemes, AAPC-HBP achieves slightly higher speedup than AAPC-RS because of better balancing in computational and communication load.

Table 3. Number of triangles in the test scenes

Scene	Number of triangles
Teapot	102080 (102K)
Balls	157440 (157K)
Lattice	235200 (235K)
Rings	343200 (343K)
Tree	425776 (426K)
Mountain	524288 (524K)

8. EXPERIMENTAL RESULTS ON A PARSYTEC CC SYSTEM

The pixel merging algorithms AAPC-HBP and ZBUF-EXC, giving the best and the worst performance results on iPSC/2 hypercube respectively, were also implemented and experimented on a Parsytec CC system. The Parsytec CC system is also a message-passing distributed-memory architecture. It contains 16 nodes, each of which is equipped with a 133 MHz PowerPC 604 processor and 64 MB memory. The interconnection network is a multistage switch network consisting of four 8×8 crossbar switching boards such that each switching board connects 4 processors to the network. The algorithms were implemented in C language and PVM 3.3 [23, 24] was used for message passing. Although hypercube topology cannot be embedded onto the interconnection topology of Parsytec for $P > 4$, a virtual hypercube topology was assumed in the implementations. As each processor has sufficiently large local memory, the algorithms were tested on relatively complex scenes selected from the publicly available SPD database [10]. The number of triangles in these scenes range from 102K to 524K. Table 3 displays the number of triangles in each scene. All results presented in this section are the timings for rendering the images given in Fig. 9 at the screen resolution of 512×512.

Figure 6 illustrates the percent decrease in the total number of pixels generated when local z-buffering is applied in the local rendering phase. As seen in Fig. 6, the percent decrease in the total number of pixels generated is very high on a small number of processors. However, it decreases with increasing number of processors for all scenes. The average percent decrease over all scenes is 49% at $P = 2$ and it reduces to 11% at $P = 16$. This is an expected result because polygons are distributed among more processors and the number of local polygons overlapping in each processor decreases. Thus, smaller number of pixels are eliminated during local z-buffering. As seen in Fig. 6 and the rendered images in Fig. 9, the percent decrease increases with increasing depth complexity of the scene. For example, the percent decrease in the *Rings* scene (Fig. 9(d)) with high depth complexity is as high as 80% at $P = 2$ and it remains above 25% at $P = 16$. Thus, local z-buffering can save a considerable amount of communication time on small to medium number of processors and for scenes containing large numbers of polygons with high depth complexity.

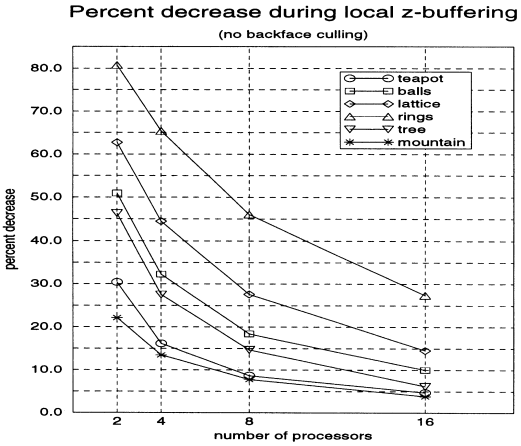


Fig. 6. Percent decrease in the total number of pixels generated after local z-buffering

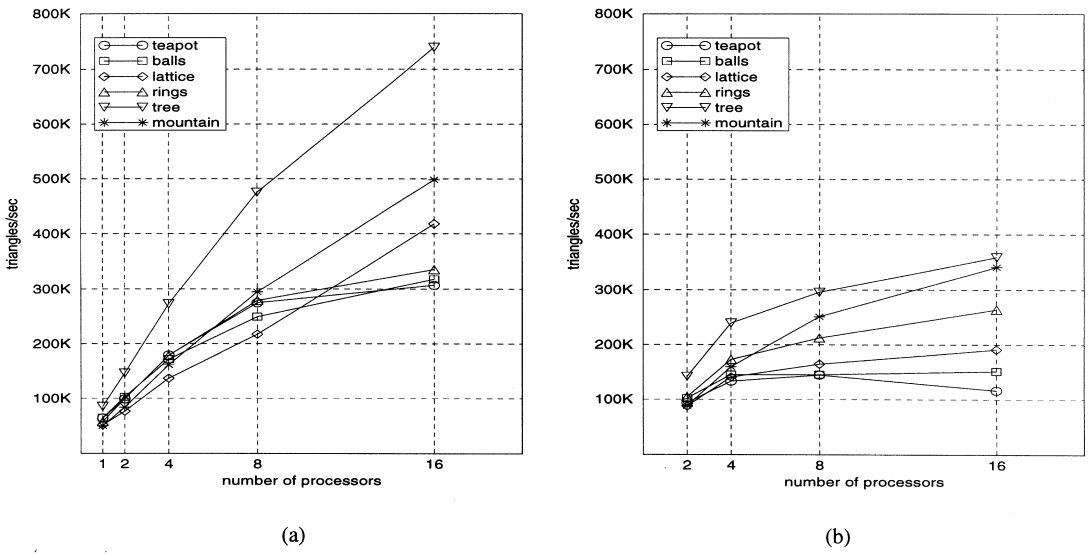


Fig. 7. Rendering rates of (a) AAPC-HBP and (b) ZBUF-EXC pixel merging algorithms on Parsytec CC system

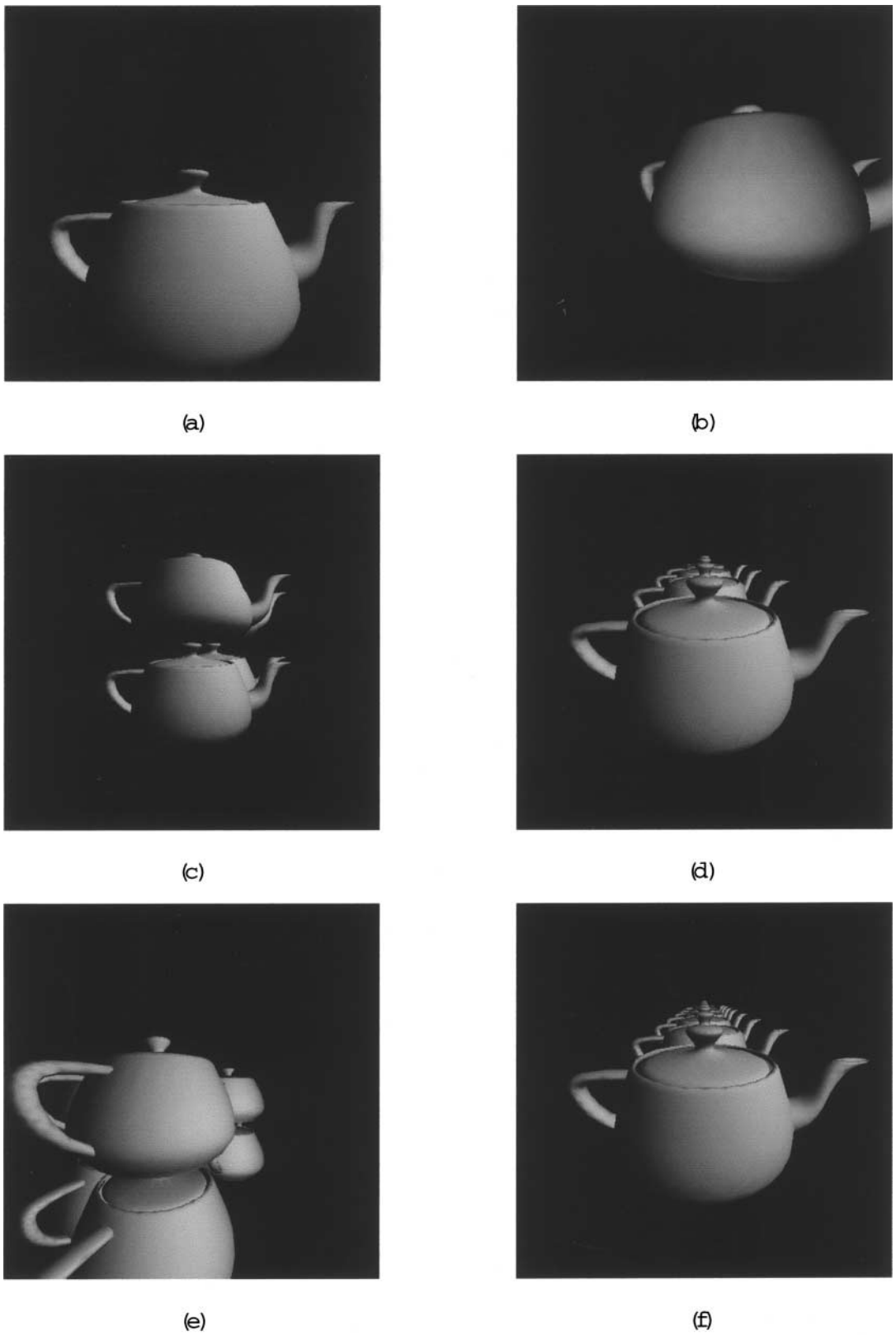


Fig. 8. Rendered images of the scenes used in the experiments on iPSC/2: (a) 1 POT scene, (b) 2 POT scene, (c) 4 POT_1 scene, (d) 4 POT_2 scene, (e) 8 POT_1 scene, (f) 8 POT_2 scene

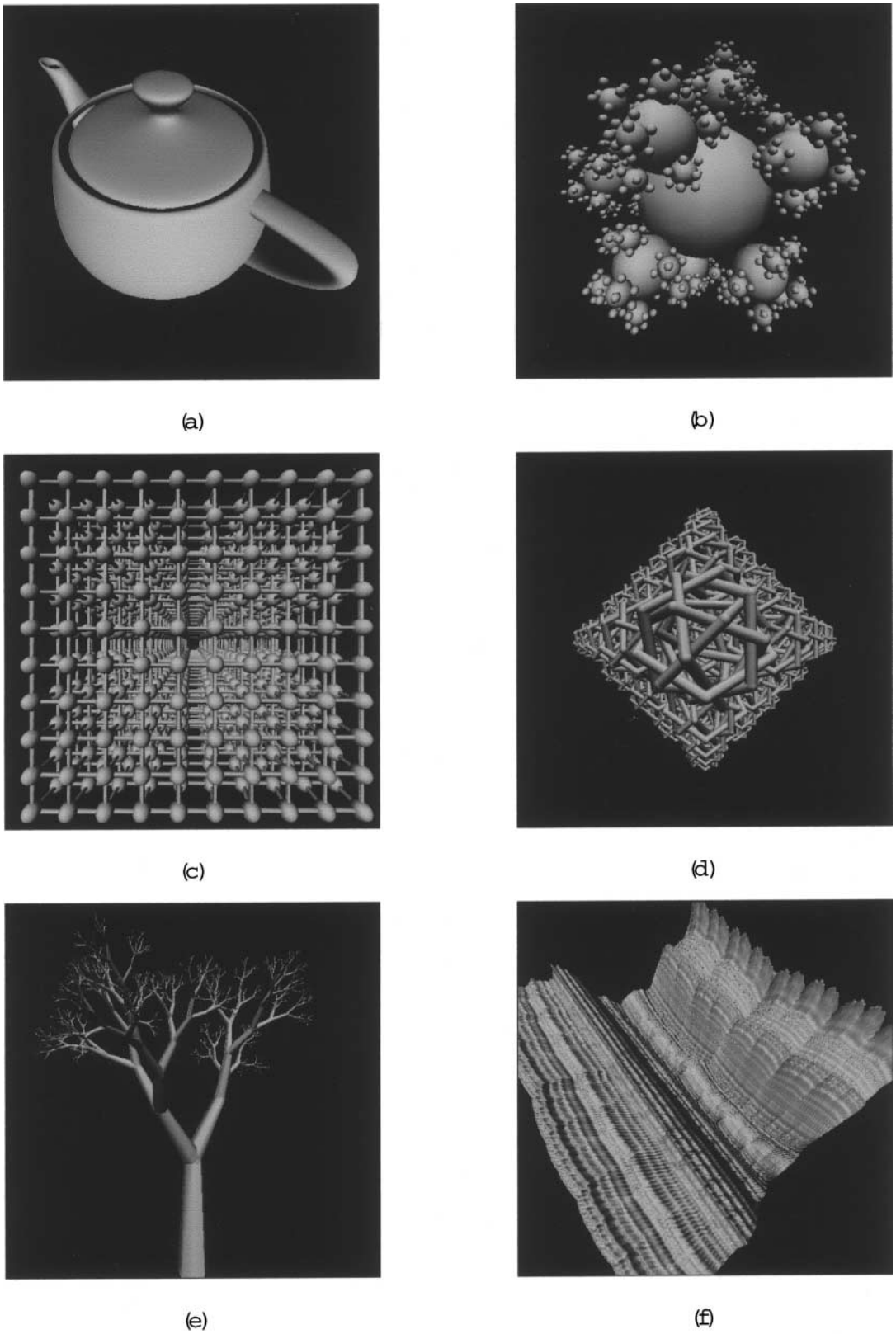


Fig. 9. Rendered images of the scenes used in the experiments on the Parsytec CC system: (a) Teapot scene (102K triangles, rendering time is 0.332 s on 16 processors), (b) Balls scene (157K triangles, rendering time is 0.495 s on 16 processors), (c) Lattice scene (235K triangles, rendering time is 0.7 s on 16 processors), (d) Rings scene (343K triangles, rendering time is 0.821 s on 16 processors), (e) Tree scene (426K triangles, rendering time is 0.576 s on 16 processors), (f) Mountain scene (524K triangles, rendering time is 1.052 s on 16 processors)

Figure 7 illustrates the variation of rendering rates of AAPC-HBP and ZBUF-EXC schemes with increasing number of processors. Rendering rate is given in terms of the number of triangles rendered per second. As seen in the figure, the AAPC-HBP scheme achieves rendering rates of 300K–700K triangles per second through speedup values of 5–10 on 16 processors. However, the ZBUF-EXC scheme can only achieve much lower rendering rates of 100K–350K triangles per second through speedup values of 2–7 on 16 processors. These results on complex scenes also verify that exchanging only active pixels results in substantial gain in the rendering rate.

The speedup values on Parsytec CC systems are lower than those on iPSC/2 systems. While PowerPC processors of Parsytec are approximately 1000 times faster than 80386/387 processors of iPSC/2 in terms of peak MFLOPS performance, the peak communication bandwidth between two nodes of Parsytec CC system (40 MBytes/sec) is only 14 times faster than that of iPSC/2 (2.8 Mbytes/sec). Hence, interprocessor communication affects the speedup performance of the algorithms more in Parsytec system than it does in iPSC/2 system. Furthermore, hypercube-specific communication schemes AAPC and pairwise exchanges used in AAPC-HBP and ZBUF-EXC respectively may incur contention on some links for $P = 8$ and $P = 16$ as hypercube topology cannot be embedded onto the interconnection topology of the Parsytec system for these P values. Such link contentions will result in the serialization of messages in the system thus increasing the communication overhead.

9. CONCLUSIONS

Efficient algorithms were proposed and implemented for object-space parallel polygon rendering on hypercube multicomputers. The proposed algorithms reduce the volume of communication by exchanging only local foremost pixels in the pixel merging phase. The proposed modified scanline z-buffer algorithm avoids message fragmentation by packing local foremost pixels in consecutive memory locations efficiently, and it eliminates the initialization of scanline z-buffer for each scanline. Several pixel merging schemes, utilizing different communication strategies and topological embeddings, were discussed for theoretical performance evaluation. Pairwise exchange and all-to-all personalized communication schemes were implemented as they were found to be best suited to the hypercube topology. All-to-all personalized communication is a direct pixel forwarding scheme, and it avoids the store-and-forward overhead of the pairwise exchange scheme at the expense of larger number of communication steps. Two adaptive screen subdivision heuristics were implemented for load balancing in the pixel merging phase. The performance of the proposed algorithms were experimented by parallel rendering of datasets from publicly available SPD database on an Intel's iPSC/2 hypercube multicomputer and a Parsytec CC multicomputer. Experimental results confirmed the expectation that active pixel merging after local z-buffering and direct pixel forwarding achieve substantial increases in the rendering performance. Rendering rates of 300K–700K triangles per second were attained in the rendering of SPD scenes containing 102K–524K triangles on 16 processors of Parsytec CC system.

The modified scanline z-buffer algorithm and load balancing heuristics proposed in this work are independent of the interconnection topology. As in hypercube topology, exchanging foremost pixels is expected to give higher rendering rates than merging full z-buffers on other topologies due to much less volumes of communication. However, the message exchange sequence of the pixel merging schemes may have to be modified to avoid link contention in the target architecture to attain maximum performance.

Acknowledgements—This work is partially supported by the Commission of the European Communities, Directorate General for Industry under contract ITDC 204–82166, and The Scientific and Technical Research Council of Turkey (TÜBITAK) under grant EEEAG–160.

REFERENCES

1. Watt, A., *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley, 1989.
2. Molnar, S., Cox, M., Ellsworth, D. and Fuchs, H., A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 1994, **14**(4), 23–32.
3. Molnar, S., Eyles, J. and Poulton, J., Pixelflow: high-speed rendering using image composition. *Computer Graphics*, 1992, **26**(2), 231–240.
4. Eyles, J., Molnar, S., Poulton, J., Greer, T., Lastra, A., England, N. and Westover, L., PixelFlow: the realization. *Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware*, Los Angeles, Aug. 1997, pp. 57-68.
5. Scopigno, R., Paoluzzi, A., Guerrini, S. and Rumolo, G., Parallel depth-merge: a paradigm for hidden surface removal. *Computers & Graphics*, 1993, **17**(5), 583–592.
6. Li, J. and Miguet, S., Z-buffer on a transputer-based machine. *Proceedings of the Sixth Distributed Memory Computing Conference*, April 1991, pp. 315-322.
7. Cox, M. and Hanrahan, P., Pixel merging for object-parallel rendering: a distributed snooping algorithm. *Proceedings of the 1993 Parallel Rendering Symposium*, Oct. 1993, pp. 49-56.
8. Lee, T. Y., Raghavendra, C. S. and Nicholas, J. B., Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 1996, **2**(3), 202–217.
9. Kumar, V., Grama, A., Gupta, A. and Karypis, G., *Introduction to Parallel Computing, Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., California, USA, 1994.

10. Haines, E., A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, 1987, **7**(11), 3–5.
11. Mueller, C., The sort-first rendering architecture for high-performance graphics. *Proceedings of 1995 Symposium on Interactive 3D Graphics*, 1995, pp. 75-84.
12. Crockett, T. W. and Orloff, T., A MIMD rendering algorithm for distributed memory architectures. *Proceedings of the 1993 Parallel Rendering Symposium*, Oct. 1993, pp. 35-42.
13. Ellsworth, D., A multicomputer polygon rendering algorithm for interactive applications. *Proceedings of the 1993 Parallel Rendering Symposium*, Oct. 1993, pp. 43-48.
14. Whitman, S., *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett Publishers, 1992.
15. Highfield, J. C. and Bez, H. E., Hidden surface elimination on parallel processors. *Computer Graphics Forum*, 1992, **11**(5), 293–307.
16. Gupta, A. and Fisher, A. L., Flexible parallel polygon rendering. *Proceedings of International Conference on Parallel Processing*, 1990, **3,** 87–91.
17. Lastra, A., Fuchs, H. and Poulton, J., Harnessing parallelism for high-performance interactive computer graphics. *Proceedings of NSF Workshop on Experimental Systems*, June 1996.
18. Onyx2, Scalable Visualization Supercomputers. http://www.sgi.com.
19. Saad, Y. and Schultz, M. H., Topological properties of hypercubes. *IEEE Transactions on Computers*, 1988, **37**(7), 867–871.
20. Nugent, S. F., The iPSC/2 direct-connect communications technology. *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, Jan. 1988, pp. 51-60.
21. Abalı, B., Özgüner, F. and Bataineh, A., Balanced parallel sort on hypercube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1993, **4**(5), 572–581.
22. Plaxton, C. G., Load balancing, selection and sorting on the hypercube. *Proceedings of 1989 ACM Symposium on Parallel Algorithms and Architectures*, 1989, pp. 64-73.
23. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V., *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.
24. Genias Software GmbH, Germany. *PowerPVM/EPX for Parsytec CC systems: PowerPVM/EPX User's Guide*, 1996.