Active Pixel Merging on Hypercube Multicomputers*

Tahsin M. Kurç, Cevdet Aykanat, and Bülent Özgüç

Dept. of Computer Engineering and Information Sci.
Bilkent University, 06533 Ankara, TURKEY

Abstract. This paper presents algorithms developed for pixel merging phase of object-space parallel polygon rendering on hypercube-connected multicomputers. These algorithms reduce volume of communication in pixel merging phase by only exchanging local foremost pixels. In order to avoid message fragmentation, local foremost pixels should be stored in consecutive memory locations. An algorithm, called modified scanline z-buffer, is proposed to store local foremost pixels efficiently. This algorithm also avoids the initialization of scanline z-buffer for each scanline on the screen. Good processor utilization is achieved by subdividing the image-space among the processors in pixel merging phase. Efficient algorithms for load balancing in the pixel merging phase are also proposed and presented. Experimental results obtained on a 16-processor Intel's iPSC/2 hypercube multicomputer are presented.

1 Introduction

There are two approaches for parallel polygon rendering in multicomputers; image-space parallelism [1, 2, 3] and object-space parallelism [4, 5, 6]. In object-space parallel rendering, input polygons are partitioned among the processors. Each processor, then, runs a sequential rendering algorithm for its local polygons. Each generated pixel is locally z-buffered to eliminate local hidden pixels. After local z-buffering, pixels generated in each processor should be globally merged, because more than one processor may produce a pixel for the same screen coordinate. The global z-buffering operations during the pixel merging phase can be considered as an overhead to the sequential rendering. Furthermore, 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 will 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 approaches in [5, 6] use architectures whose processors are interconnected in a tree structure for pixel merging phase. Both approaches result in low processor utilization in pixel merging phase due to tree topology. The processors in the lower levels of the tree (e.g., processors at the leaves) may have substantially less work than those in the upper levels of the tree. Another approach presented in [4] utilizes network broadcast capability for pixel merging phase. Each processor, starting from the first processor and continuing in increasing processor id, broadcasts "active" pixels to a global frame buffer. The other processors capture the broadcast pixels and delete their local pixels which are hidden by the broadcast pixels. In this way, the number of pixels broadcast by the next processor is expected to decrease. Their approach will introduce a large communication overhead due to broadcast operation on medium-to-coarse grain distributed-memory architectures. In addition, their approach suffers from low processor utilization because a processor remains idle until the end of pixel merging phase after broadcasting its pixels.

This paper investigates the object-space parallelism on hypercube-connected distributed-memory multicomputers. In our approach, the hypercube interconnection topology and message passing characteristics of hypercube multicomputer are exploited. Algorithms proposed in this work achieve good processor utilization by implicitly subdividing image-space among the processors in pixel merging phase. The volume of communication is decreased by only exchanging local foremost pixels for active pixel locations as in [4]. However, storing only local foremost pixels for efficient pixel merging introduces some overhead to conventional scanline z-buffer algorithm. An algorithm, called modified scanline z-buffer, is proposed to reduce this overhead. The proposed algorithm also avoids initialization of scanline z-buffer for each scanline in local z-buffering. Load balancing issue in pixel merging phase is discussed. Algorithms for achieving better load balance are proposed and discussed.

2 Modified Scanline Z-buffer Algorithm

In order to prevent message fragmentation in active pixel merging, the local foremost pixels should be stored in consecutive memory locations. In this section, a modified scanline z-buffer algorithm is presented. This algorithm utilizes a modified scanline scheme to store foremost pixels in consecutive memory locations efficiently. In addition, this algorithm avoids initialization of scanline z-buffer for each scanline by sorting polygon spans at each scanline in increasing minimum x-intersections.

When polygons are projected to the screen (of resolution NxN), some of the scanlines intersect the edges of the projected polygons. Each pair of such intersections is called a *span*. In the first step of the algorithm, the *spans* are generated and put into the *scanline span lists*. The *scanline span lists* involve a linked list for each scanline which contains the respective polygon spans. 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. Scanline 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_{min} values. This sorting allows to perform local z-buffering without initializing the scanline array for each scanline on the screen.

In the second step, spans in the scanline lists are processed, in scanline order (y order), for local z-buffering and shading. Two local arrays are used to store only local foremost pixels. First array is called *Winning Pixel Array* (WPA) used to store 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 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 *Modified Scanline Array* (MSA) of size N, is a modified scanline z-buffer. MSA[x] gives the index in WPA of pixel generated at location x. Initially, 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 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 sorting of spans in scanline span list avoid re-initialization of MSA at each scanline. However, due to comparison made with "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 scanline span lists 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 previous spans. The current span s is divided into two segments such that one of the segments cover the pixels generated by previous spans in the current scanline and other segment covers the pixels generated by spans of previous scanline. Distance comparisons are made for the pixels in the first segment. The pixels generated for the second segment are stored into WPA without any distance comparisons.

3 Pixel Merging on Hypercube Multicomputer

This section presents two active 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 NxN. Then, a global z-buffering operation is performed on local foremost pixels so that each processor gathers global foremost pixels belonging to a horizontal screen subregion of size NxN/P.

3.1 Pairwise Exchange Scheme

This scheme exploits the *recursive-halving* idea widely used in hypercube-specific global operations. This operation requires d concurrent divide-and-exchange stages. Within 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 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 over 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.

3.2 All-to-All Personalized Communication Scheme

The pairwise exchange scheme can also be considered as a *store-and-forward* scheme. At each stage, the received pixels are stored into the local memory of the processor. These pixels are compared and merged with the pixels retained. After this merge operation, some of the pixels are sent at the next exchange stage, i.e., they are forwarded towards the destination processor through other processors at each concurrent communication step. Note that during these store-compare-and-forward stages, 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 their destination processors.

In iPSC/2 hypercube multicomputer, communication between processors is done by Direct Connect Modules (DCMs). Communication between two non-neighboring processors is almost as fast as 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 [7]. Using DCMs, we can exchange messages between non-neighbor processors by the algorithm presented in [8]. This algorithm totally avoids message congestion by ensuring that at each exchange stage, the pixel data is directed to destination processors following disjoint paths.

In all-to-all personalized communication scheme, the screen is implicitly divided into P horizontal subregions. Each subregion is implicitly assigned to a processor. Then, each processor sends the pixels belonging to the subregion of processor "k" directly to processor "k". After $P-1$ exchange steps, each processor z-buffers the local pixels with the received pixels. Each processor holds a local z-buffer of size $N \times N/P$. 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 pixels are processed, z-buffer contains the pixels in the final picture.

4 Load Balancing in Pixel Merging Step

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

4.1 Recursive Adaptive Subdivision

This scheme recursively divides the screen into two subregions such that number of pixels in one subregion is almost equal to the number of pixels in the other subregion. This scheme is well suited to the recursive structure of the hypercube.

Each processor counts the number of local foremost pixels at each scanline and stores them in an array. Each entry of the array stores the sum of local foremost pixels at the corresponding scanline. An element-by-element global prefix sum operation is performed on this array to obtain the distribution of foremost pixels in all processors. Then, using this array, each processor divides the screen into two horizontal bands of consecutive scanlines so that each region contains 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 bottom subregion is assigned to other subcube. Subcubes perform subdivision of the local subregions concurrently and independently. Since screen is divided into horizontal bands, the global array obtained by global sum operation is used for further divisions of the screen.

4.2 Heuristic Bin Packing

In the recursive adaptive subdivision scheme, the subdivision 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 subregion. In addition, when a division point is found and screen is divided into two subregions, each subregion is subdivided independent of the other one. As a result, at each recursive subdivision, the load imbalance between the subregions may propagate and increase. Therefore, at the end of recursive subdivision, some processors may still have substantially more work load than others. A better distribution of work load among the processors 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.

5 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 and 640x640. The characteristics of the scenes are given in Table 1. The abbreviations in the figures and tables are AAPC: *all-to-all personalized communication*, PAIR: *pairwise exchange*, RS: *recursive adaptive subdivision*, HBP: *heuristic bin packing*, ZBUF-EXC: *full z-buffer merging*. All timing results in the tables are in milliseconds.

Table 2 illustrates the performance comparison of PAIR-RS scheme with full z-buffer merging. The timings for some scene instances for ZBUF-EXC scheme

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 (TPF) for different screen sizes.

Scene	Num. Of Polygons	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. Relative execution times of full z-buffer merging 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	*	*	*

could not be obtained due to insufficient local memory. Those cases are indicated by a "*" in this table. As seen in Table 2, PAIR-RS gives much better results than ZBUF-EXC in pixel merging phase. Since pixel information for inactive pixel locations are also exchanged, the volume of communication in ZBUF-EXC is larger than that of PAIR-RS. As is also seen from the table, the PAIR-RS performs better than ZBUF-EXC also in local z-buffer phase since it avoids initialization of z-buffer.

Total volume of concurrent communication (in bytes) for various pixel merging schemes are illustrated in Fig. 1. The total volume of concurrent communication is calculated as the sum of the maximum volume of communication at each communication step. As seen from the figure, AAPC scheme results in less volume of communication than PAIR scheme as expected. Note that the volume of communication 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, it is expected that volume of communication decreases as the number of processors increases as is also seen in Fig. 1(a). The increase in volume of communication
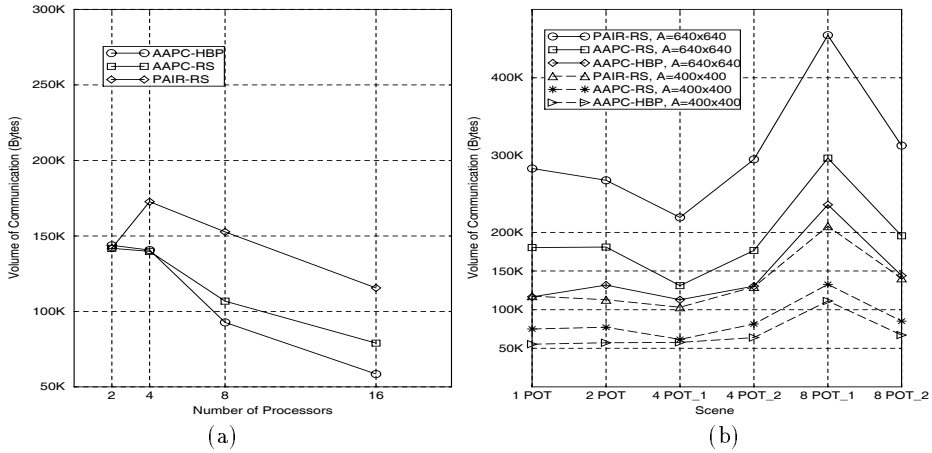
Fig. 1. Volume of communication for (a) 2 POT scene on different processors, $A = 400 \times 400$. (b) $A = 400 \times 400$ and $A = 640 \times 640$ for different scenes on 16 processors.

in PAIR-RS scheme on 4 processors is due to store-and-forward overheads. It is also experimentally observed that better load balance in pixel merging indirectly affects the volume of communication as well. As illustrated in Fig. 1(b), HBP scheme results in less volume of communication than RS scheme.

Performance comparison of load balancing heuristics are illustrated in Fig. 2. The *load imbalance* is the ratio of the difference of the work loads of maximum and minimum loaded processors to average work load. The work load of a processor was taken to be the number of pixel merging operations it performs in the pixel merging phase. As seen from the figure, HBP achieves much better load balance than RS as expected. Load balance improves with increasing screen resolution due to better accuracy in dividing the screen. As is also seen from Fig. 2(a), HBP scales better than RS for larger number of processors. A speedup of 11.47 was obtained using 16 processors with AAPC-HBP scheme for 2 POT scene and $A = 640 \times 640$.

6 Conclusions

In this work, efficient algorithms were proposed for active pixel merging on hypercube multicomputers. These algorithms reduce the volume of communication by exchanging only active pixel locations in pixel merging phase. The message fragmentation in active pixel merging is avoided by storing local foremost pixels to consecutive memory locations in local z-buffering phase. An algorithm, called modified scanline z-buffer, is proposed to store the local foremost pixels into consecutive memory locations efficiently. This algorithm also avoids initialization of scanline z-buffer for each scanline on the screen. It is experimentally observed that active pixel merging with modified scanline z-buffer algorithm performs better than full z-buffer merging. It is also experimentally observed that *all-to-all personalized communication* scheme achieves less communication overhead than *pairwise exchange* scheme due to less store-and-forward overheads in active
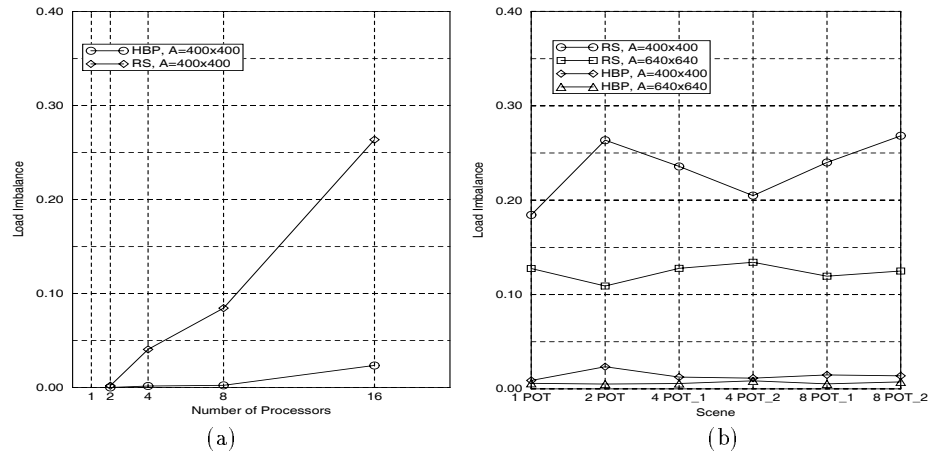
Fig. 2. Comparison of RS with HBP. (a) Different number of processors for 2 POT scene, $A = 400 \times 400$. (b) Different screen resolutions and different scenes on 16 processors.

pixel merging. Two load balancing heuristics were proposed to distribute load evenly in pixel merging. The *heuristic bin packing* achieves better load balance and scales better than *recursive adaptive subdivision* in active pixel merging. Therefore, it is recommended that *all-to-all personalized communication* with *heuristic bin packing* scheme should be utilized for active pixel merging on hypercube multicomputers.

References

1. J.C. Highfield and H.E. Bez, 'Hidden surface elimination on parallel processors', *Computer Graphics Forum*, **11**(5) 293-307 (1992).
2. D. Ellsworth, 'A multicomputer polygon rendering algorithm for interactive applications', in *Proc. of 1993 Parallel Rendering Symposium*, San Jose, 43-48 (Oct. 1993).
3. S. Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers, Boston (1992).
4. 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 (Oct. 1993).
5. 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).
6. J. Li and S. Miguet, 'Z-buffer on a transputer-based machine', in *Proc. of the Sixth Distributed Memory Computing Conf.*, IEEE Computer Society Press, 315-322 (April 1991).
7. S.F. Nugent, 'The iPSC/2 direct-connect communications technology', in *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.*, 51-60 (Jan. 1988).
8. 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).