Full-Frame Merging for Sort-Last Polygon Rendering on a Multicomputer

J. M. Pereira[12], C.A. Wüthrich[13] and M. R. Gomes[12]

[1] INESC, Grupo Computação Gráfica, R. Alves Redol 7, 1000 Lisboa, Portugal

[2] IST, Instituto Superior Técnico, R. Rovisco Pais, 1000 Lisboa, Portugal

[3] Dept. of Informatics and Mathematics, School of Architecture and Civil Engineering (HAB), Coudraystr. 13, D-99423 Weimar, Germany.

Abstract

We propose a refinement of the Sort-Last algorithms' classification based on the scheduling of the rendering and merging steps.

Two algorithms whose rendering and merging steps run consecutively are described. Two different approaches were taken to implement the merging step: the Distributed Framebuffer approach and the Pipeline Composition approach. The load balancing problem is also discussed: a dynamic request-based mechanism is implemented at the end of the rendering phase.

Another solution, the ScanlineFlow Rasterization algorithm, is described. Its main characteristic resides on the fact that both steps, rendering and merging, run concurrently. This solution has provided good results and is a viable alternative to implement sort-last algorithms on a multicomputer.

The three algorithms made use of the full-frame merging technique because merging a full frame from each node is very regular and easy to implement.

Our developing platform consisted of a Parsytec MultiCluster machine with sixteen processors running the Helios Operating System and using the CDL (Component Distribution Language) parallel programming language.

1. Introduction

As is well-known [Molnar90], there are two main phases in the rendering process that account for most of the computation time: the geometric transformation phase and the rasterization phase. We ignore in the rendering process the problem of traversing the object database prior to rendering. Recently, the task of classifying parallel graphics algorithms has been improved by the proposal of a new conceptual model [Molnar94] for multiprocessor architectures that looks at parallel rendering as a sorting problem. The basis for this taxonomy derives from the fact that the rendering can be viewed as a problem of sorting primitives to the screen, as noted by [Sutherland74]. For fully parallel renderers (systems where both geometry processing and rasterization are performed in parallel), this sorting involves a redistribution of data between processors, because responsibility for primitives and pixels are distributed. The sort can, in general, occur anywhere in the rendering pipeline: during geometry

processing (sort-first), between geometry processing and rasterization (sort-middle), or during rasterization (sort-last). Sort-first algorithms redistribute "raw" primitives (before their screen-space parameters are known), sort-middle redistribute screen-space primitives and sort-last redistributes pixels (pixel fragments or samples). Sort-last (SL) can be done in two ways. One approach, called SL-*sparse*, minimizes communication by distributing only those pixels actually produced by scan-conversion. The second approach, called SL-*full*, stores and transfers a full image from each renderer.

This paper proposes a more subtle classification for sort-last algorithms based on the fact that compositing (or pixel merging) can be performed during the scan-conversion or after the scan-conversion. Three SL-full algorithms are described. The new classification allows to highlight the differences between them and so, proceed easily to comparison and analysis.

We have implemented two approaches where the merging operation occurs after pixel rendering. In the first approach, Distributed Framebuffer, the partially rendered frame buffers are merged into distributed image space frame buffers which are finally copied into the real frame buffer of the machine. The second one, Pipeline Composition, taking advantage of the pipeline interconnection network, we used an algorithm based on the image-compositing scheme of the PixelFlow machine [Molnar 92]. Upon completion, each node must execute three distinct tasks: receiving the frame buffer from the previous node, z-buffering each of its own pixels with each of the pixels it receives from the previous node and streaming the resulting pixels to the next node. Our implementation of this algorithm has a unique feature in the sense that these three operations are multithreaded, increasing this way the system's throughput.

A solution where pixel merging operation occurs during the scan-conversion was also implemented: the ScanlineFlow Rasterization algorithm. By taking, again, advantage of a pipeline interconnection network and since this algorithm renders an image one scanline at a time, each node rasterizes multiple polygons active on a given scanline after synchronization with the previous node in order to read the rasterization data concerning that scanline. The execution is pipelined, in the sense that while a node is rasterizing a scanline, it is also receiving the next scanline from the previous node and it is sending the previous scanline (already rasterized) to the next node. Like Pipeline Composition algorithm, these three operations are multithreaded.

The load balancing problem is also discussed. In fact, it has been claimed in the literature that object-parallel processing in sort-last should "naturally load balance" [Cox92] or that "load balancing is automatic" [Molnar91]: in the straightforward implementation of SL approach the primitives are shuffled to the processors in a round-robin fashion. No corroboration of the claim has been offered, either analytically or experimentally, so far. Thus, when dynamic load imbalances may not be avoided, a dynamic request-based scheme was employed at the Distributed Frame buffer and the Pipeline Composition algorithms.

We designed our sort-last rendering algorithms to map onto a class of general purpose parallel architectures, specifically, MIMD distributed memory (message-passing) systems keeping in mind the following idea: new low-cost parallel machines have recently appeared or are under development

promising a much improved network communication bandwidth and speeds, which means that there is room for a new analysis taking into account the new communication speeds and the actual communication requirements of this class of algorithms that fortunately can be estimated with some accuracy.

2. Sort-Last Algorithms Classification

First we must define precisely some terms. To do that we will consider that the rasterization step of the graphics pipeline has been broken into two stages called *pixel rendering* (computing pixel values) and *pixel merging* (determining which pixels are visible). We classify an algorithm as sort-last since it redistributes pixels (or pixel fragments, or samples) over the interconnection network. This redistribution can occur **after** each node has finished to generate pixels for its subset of the graphics database, which means that merging will take place after the pixel rendering, or it can occur **as** each node generates pixels, which means that merging takes place during the pixel rendering.

As said before, the convention of [Molnar94] proposes two types of sort-last algorithms, SL-full and SL-sparse, which make use of the concept of *active* and *inactive* pixels at each processor. We say that a pixel location is *active* if at least one pixel has been rendered to that position, and that it is *inactive* otherwise. SL-sparse algorithms merge only active pixels. SL-full algorithms merge a full frame (all active and inactive pixel locations) from every rendering processor. Full-frame merging takes advantage of the fact that merging a full-frame from each processor is very regular and, thus, can be implemented easily.

We propose a refinement of this classification based on the scheduling of the rasterization stages. In fact, another variable is the specification of when the pixel rendering/merging stages are to run: are they run concurrently or consecutively? The former case is designated by Sort-Last *Merging-First* (**SLMF**) where computation and communication are overlapped. The later one is called Sort-Last *Merging-Last* (**SLML**). Again, these two sorts of algorithms make use of either the sparse merging or the full-frame merging techniques.

Let's see how known SL systems are placed in the context described above.

In the most straightforward solution to SL rendering on a multiprocessor, each node is assigned responsibility for merging pixels from some subset of the screen resolution. There are two obvious ways of implementing the pixel merging. In the first one, as each node generates pixels for its subset of the graphics database, it sends each pixel to the destination node that is responsible for merging for that pixel location. This is typically a SLMF-sparse algorithm and was explored in commercial systems like [Evans92, Fuji93, Kubota93] and in software systems [Cox95]. In the other way, each processor renders its subset performing local z-buffer (that is, with respect to the other pixels it generates) and producing pixels into a local frame buffer. Then, the active pixels from all nodes' local frame buffers are merged into the global frame buffer. We are facing a SLML-sparse algorithm. This solution implies the need of local frame- and z-buffers between the processors and pixel merging

network. It has been shown experimentally that, despite under utilization of the local z-buffering hardware, traffic savings of about 20%, when compared with the above method, should be expected [Cox95]. An alternative SLML-sparse algorithm, when network broadcast is available, is the Distributed-Snooping merge algorithm [Cox93].

In the PixelFlow algorithm [Molnar91], processors are connected by a pipeline network. Upon rendering completion each node streams its full frame to the next node. In the end, the last node contains a correctly z-buffered image. This is a SLML-full algorithm, which is currently under construction at the University of North Carolina, Chapel Hill [Molnar92].

As far as we know, no SLMF-full scheme has been proposed so far. We will present in this paper the ScanlineFlow Rasterization algorithm that should be included in the SLMF-full category.

3. The MultiCluster System Overview

MultiCluster2 from Parsytec is a MIMD message-passing system (also known as a multicomputer) with sixteen Transputers T800, each processor with 4 Mbytes of private memory, and a reconfigurable network. It runs the Helios Operating System. With a user interface similar to that provided by Unix, Helios is designed to work on transputers that have no memory management hardware. Helios provides a facility called multi-tasking which enables multiple tasks to be run on one or more processors.

The Component Distribution Language, or CDL, enables a programmer to carry out parallel programming under Helios. The purpose of CDL is to provide a high-level approach to parallel programming, where the programmer defines the program components and their relative interconnections (the application topology) independently of the size and topology of the Transputer network. It is, then, Helios who is responsible for mapping the task force onto the available physical resources.

4. Algorithms Description

The SL algorithms explore object-space parallelism: graphics primitives are assigned to processors disregarding their screen location, and each processor completely renders the primitives it is assigned. The sort from object coordinates to screen coordinates (i.e. the pixel merging) takes place 1) after individual images (of part of the dataset) have been rendered (SLML-full) or 2) during the rendering (SLMF-full).

Since we are using a message-passing commercial architecture the basic software model for the implementation of the algorithms consists of two major components: the central controller and the nodes. We have implemented symmetrical algorithms in which each node serves both as a renderer and as a compositor.

The central controller reads in the polygonal object descriptions from disk files and then distributes them to the nodes by using the *scattering* method [Molnar 91]. If there are P primitives and N nodes we simply assign P/N primitives to each node. This assignment is done by shuffling primitives in a round-robin fashion, that is assigning the first primitive to the first node, the second to the second node, and so forth. The primitives in most databases contain some amount of geometric coherence. That is, primitives near each other in the database file, generally lie near to each other in the image as well. This means that scattering distributes coherence among the nodes; in other words, scattering distributes nearby primitives over each of the nodes. So, the scattering method tries to minimize two sources of static load imbalances: unequal numbers of primitives on the nodes and unequal rasterization times due to the size and shape of the primitives.

4.1 - SLML-full Methods

Each node applies to the incoming polygons the whole graphics visualization pipeline.

We implemented a polygon scan-conversion algorithm under the assumption that all polygons of the database are triangles. The scan-conversion algorithm incorporates the Z-Buffer procedure for hidden-surface removal along with a smooth-interpolation shading scheme.

It is trivial to partition the database into parts having an equal number of primitives by using scattering as mentioned above. However, these parts can be of widely differing sizes; in this case, scattering may not be good enough to statically load balance the processing times on each node. Hence, some nodes may complete processing sooner than others. In this case, a dynamic balancing load scheme is needed to transfer work from nodes that are still busy to idle nodes.

Our strategy works as follows: as soon as a node finish computing its initial database, it sends a request for a *work packet* (a set of polygons) from a node still working on its own database. This request moves along a ring. If the neighbour has available work, it splits the queue of polygons in two parts and responds with a packet of polygons; if not, the request is routed to next node. If the request returns without satisfaction, the node concludes that all the other local databases have been computed. This is the local termination detection implemented and is sufficient for our application. No load information need to be exchanged. The code that implements the dynamic load balancing scheme runs concurrently with the main task (rasterization procedure).

To ensure a balanced load, two parameters must be determined in order to satisfy two rules: a reasonable work splitting mechanism and the cost of communicating a work packet must be significantly smaller than the ultimate time cost of executing the work packet in place. The two parameters are related and concern to the size of the work packet. The first parameter specifies the local database splitting factor and determines the number of polygons a work packet must contain. The second parameter fixes the minimum number of polygons of a work packet. The determination of these parameters is heuristic and difficult to achieve, since we chose not to estimate computation times during rendering, but to use a simpler decision criterion based on the number of polygons to be processed. [Pereira95] gives us some hints to assign reasonable values for these parameters.

When all nodes have detected their local termination (each node has sent a request packet that traveled along the ring without satisfaction) pixel rendering has finished and the local frame buffers are then ready to be merged. Two approaches were taken to implement this phase: Distributed Frame buffer and Pipeline Composition [Pereira95].

In the Distributed Frame buffer (**DF**) approach each node is made responsible for a specific area of the screen (image space partition); when the merging phase starts it asks to each other of (N-1) nodes for the portion of the frame buffer concerning that screen region and performs the depth-comparison on it. This procedure is performed simultaneously by all nodes, involving this way an all-to-all communication scheme. However, it is very difficult to program in CDL this form of connectivity between all nodes because it does not scale very well. In fact, as the number of nodes grows, the number of I/O streams to be allocated would increase considerably. To avoid this situation, we should implement a topology where the number of I/O channels was fixed whatever the number of nodes of the system. A 2D torus topology was chosen based on the fact that such a topology is easily scalable, since each node has always the same number of links and recent studies have been conducted on their communication efficiency [Badouel92]. As routing algorithm we implemented the so-called *e-cube* algorithm [Dally87].

The Pipeline Composition (**PC**) approach was based on the PixelFlow algorithm [Molnar91]. In the straightforward implementation of this approach, each node receives the frame buffer of the previous neighbour, Z-buffers its contents with its own frame buffer and then sends the resulting frame buffer to next node. Immediately, we conclude that this approach is not efficient since only one node is activated. In order to exploit as much as possible the available parallelism of our platform, instead of considering a frame buffer as whole task, we divided it into small chunks, scan-lines, and pipelined these. With this solution, all nodes are performing merging simultaneously. Besides that, taking advantage from the fact that communication and computation can be overlapped on a Transputer, we have been able to overlap, in each node, the three operations needed to perform the merging: reading, Z-buffering and sending. Each node has an input double-buffer into which it holds information of two scan-lines: while one buffer is being accessed by the Z-buffer thread to perform the depth-comparisons with the local scan-line, the other buffer is available to the read thread for receiving the next scan-line from the previous node. After finishing the z-buffering operation, the node can start immediately with the next scan-line by swapping the buffers. Meanwhile, the send thread is accessing the frame buffer to transmit the scan-line that has been recently processed and the read thread is reading the next scan-line. The control and management of the input double-buffer and the frame buffer are done using of semaphores. The first node only needs to execute the send thread, and the last node only executes the reading and Z-buffer threads.

The Distributed Frame buffer scheme requires that, after merging, each node sends its partition of the screen to the central controller in order to build the final frame buffer. Concerning the Pipeline Composition approach, we don't need to perform any frame buffer communication to the central

controller because the frame buffer of the last node of the composition network contains, after merging, the whole picture information.

4.2 - SLMF-full Method

We will describe now the ScanlineFlow Rasterization algorithm which assumes a pipeline interconnection network for its execution. The implementation of this strategy was based on the Scanline z-Buffer sequential algorithm [Rogers85].

In comparison with z-buffer algorithm, where the state information necessary for rendering a pixel is stored for every pixel on the screen, a scanline rendering algorithm presorts the object database in screen space, and renders each scanline individually - only one scanline of pixel state information is kept. This is a two pass algorithm. In the first pass, the polygons are transformed, shaded and bucket sorted by the number of the first scanline on which they first become active. Then, in the second pass, the bucket sorted list is traversed in screen-y order, maintaining an active polygons list which are, then rasterized.

After the polygon distribution by the scattering method, all processors perform the first pass of the algorithm. The rasterization for each scanline starts after the polygons have been sorted. Each node rasterizes multiple polygons active on a given scanline after synchronization with the previous node in order to receive the rasterization data concerning that scanline (active and inactive pixels). The execution of the algorithm through the system is pipelined: while a node is processing scanline y, the previous node is processing scanline y+1 and the next node is processing scanline y-1. The last node generates scanlines with the correct information. Communication and computation are overlapped in the sense that while a node is rasterizing a scanline, it is also reading the next scanline from the previous node and forwarding the previous scanline to the next node. Each processor has a triple-buffer into which it holds information of three scanlines and its control and management is done by using semaphores. This scheme of information flow is very similar to the PC algorithm. The strongest point of this solution resides on the fact that depth-comparison is performed once for each scanline but this comes at the expense of synchronization that affects the throughput and the latency of the algorithm.

5. Performance Results

Since the rasterization phase of the graphics pipeline is the most time consuming section, our attention had focused on it. The performance tests don't consider the time spent on geometric transformations. We are finishing a version where a complete renderer is implemented in each node.

We assume that we are given ASCII files containing polygonal representations of 3D objects in screen space coordinates with backfaces culling performed. Another simplification made by us impose that the objects are described as collections of triangular facets. The input scenes are rendered with a resolution of 512 x 512, producing graphics images in Portable Pixel Map (ppm) format.

The execution times of sequential versions of the z-buffer triangle rendering and scanline z-buffer rendering algorithms **at one processor** are shown. Several test scenes, illustrated in Appendix A, were used:

Images	Number of triangles (screen coordinates)	Uniprocessor Execution Time (secs)	
		z-buffer	scanline z-buffer
Teapot	11666	12.68	18.62
Gears	16207	31.8	65.62
Misc_I	16727	18.0	28.87
Misc_II	25661	17.57	37.2
Tetra	34017	17.180	52.78
Misc_III	48168	30.0	97.89
Mountain	75733	40.48	

The Teapot, Gears, Tetra and Mountain scenes were created by using the well-known Standard Procedural Databases (SPD) from Eric Haines [Haines87].

The timing facilities provided by Helios are not very accurate since the clock resolution is only 10 ms.

5.1 - Dynamic load balancing experiments

Load balancing is the primary focus of most designers of parallel programs. In measuring the contribution of load imbalance, the finishing times of the rendering stage of each node are noted. The tests consisted of determining the usefulness of the request-based dynamic load-balancing scheme. To do that, we ran the PC algorithm with the dynamic load balancing feature enabled and disabled, and registered the load imbalance percentages produced in both situations. The *load imbalance percentage overhead* is calculated this way: the difference in time between the longest time taken by a node to perform pixel rendering and the average of all nodes is recorded and then divided by the longest time value used previously. However, since we are working with finishing times, a low load imbalance percentage overhead (all nodes have similar finishing times) doesn't mean better performance results because total execution time can be larger due to the time spent on the load balancing algorithm. A measure of performance, like Throughput (polygons/second), must also be considered to check of the effectiveness of the solution.

Table 1 shows results obtained for some scenes.

Gears Image					Misc_I Image			
Number of Nodes	Dynamic Load Balancing enabled		Dynamic Load Balancing disabled		Dynamic Load Balancing enabled		Dynamic Load Balancing disabled	
	Load Imbalance (%)	Throughput (pol/sec)	Load Imbalance (%)	Throughput (pol/sec)	Load Imbalance (%)	Throughput (pol/sec)	Load Imbalance (%)	Throughput (pol/sec)
2	0.8	730	6.8	687	1.3	1131	0.8	1137
4	0.6	1036	18.7	956	0.6	1463	24	1378
8	4.4	1289	38.2	1138	25.1	1570	48.4	1468
16	11.4	1456	57.1	1296	51.6	1578	68.8	1561
Misc_III Image					Mountain Image			
Number of Nodes	Dynamic Load Balancing enabled		Dynamic Load Balancing disabled		Dynamic Load Balancing enabled		Dynamic Load Balancing disabled	
	Load Imbalance (%)	Throughput (pol/sec)	Load Imbalance (%)	Throughput (pol/sec)	Load Imbalance (%)	Throughput (pol/sec)	Load Imbalance (%)	Throughput (pol/sec)
2	0.9	2160	0.1	2158	0.7	2625	0.1	2625
4	0.9	3224	0.1	3158	1.2	3948	7.6	3807
8	7.5	4366	0.3	4464	7.8	5597	19.2	5426
16	10.2	5400	1	5542	8.2	6773	34.6	6598

Table 1 - Load Imbalance percentages and Throughputs

A first analysis of table 1 shows that some improvements were achieved mainly for the Gears and Mountain images. With Misc_III image, since it contains similar and very small triangles, the load balancing algorithm is an overhead. Facing these figures, we can conclude that only for images that contain a few very large primitives (like the floor on the Gears image or the ground on the Mountain image), scattering polygons would not be sufficient to achieve good load computational loads, and the adding of an extra level of load balancing scheme (a dynamic one) would be useful. Thus, it seems that, in general, scattering would be enough. However, such conclusion may be too quickly drawn because several factors that tend to unbalance the workload were not considered in our first versions of the programs. First, the culling and clipping steps require a different number of operations for different triangles. Second, the polygons to be rasterized may not be triangles and so they may be of vastly differing shapes and sizes. A definitive conclusion about the need of an extra-load balancing scheme in SL algorithms can not be given: further research is required.

5.2 - Algorithms Performance Evaluation

The request-based mechanism in the SLML algorithms was activated only for the test cases where the performance has improved (like the Gears or the Mountain scenes).

The usual measure of the effectiveness of a parallel algorithm is speedup, defined as the time to execute a problem on a single processor divided by the time to execute it on N processors. Efficiency is also a useful measure that gives us an indication of the utilization of the processors in the system, and can be obtained by dividing the speedup by the number of used processors.

Concerning the SLML-full strategy, the total execution time of both algorithms, PC and DF, can be broken into a perfectly parallel component, pixel rendering, and an explicit overhead component, the merging phase. In fact, the rendering phase of both programs speeds up linearly as the number of nodes increases. The merging phase includes computations that do not exhibit linear speedup and communications. In the PC algorithm, the merging phase time suffers from minor variations as the number of nodes increases: they are due to the pipeline latency. On the other hand, the DF approach exhibits a merging phase that changes a lot as the number of nodes varies due mainly to communication costs. The communication costs associated to the DF algorithm are very high. While PC approach uses a very simple structure for its topology, the DF strategy uses a 2D torus, which means that the mapping of this later topology onto the physical network by the Helios operating system has much more communication costs associated than mapping a pipeline [Pereira95]. Thus, the speedup results show that PC algorithm performs much better than DF algorithm as can be noted in the figures 1 and 2.

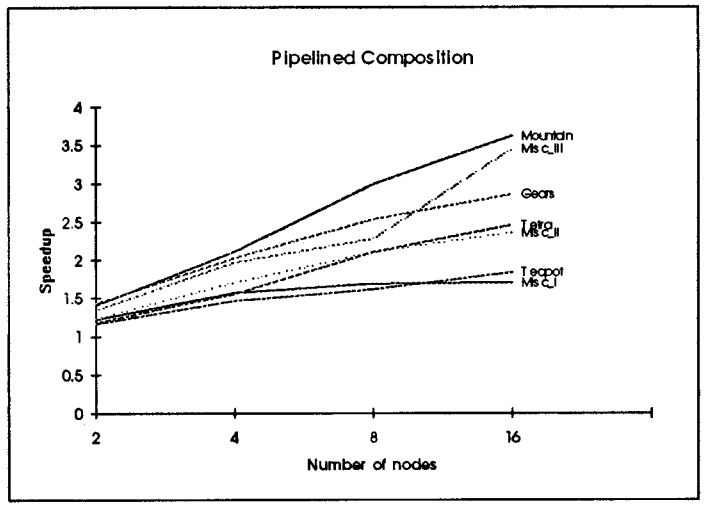

Figure 1 - Speedup results for the PC algorithm

In the PC algorithm, speedup performance continues, in general, to increase as processors are added, even for the smallest scene, although large number of processors are most effective for more complex scenes: speedup becomes gradually worse primarily due to the fact that the percentage of the total time execution taken by the merging phase increases as nodes are added.

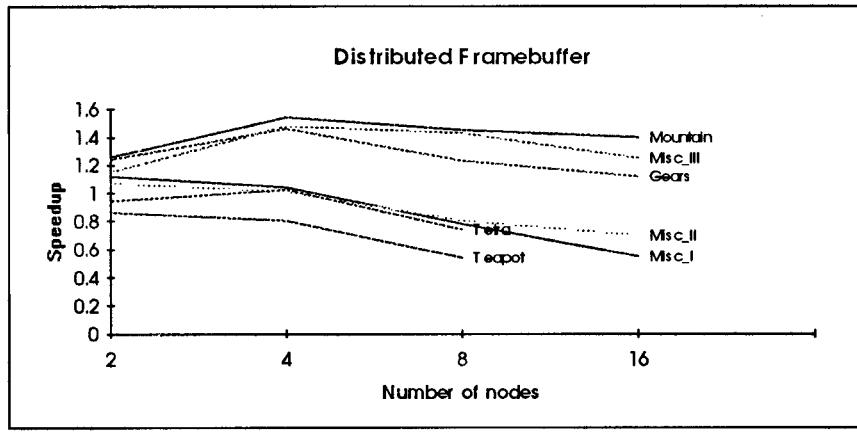

Figure 2 - Speedup results for the DF algorithm

In the DF algorithm the results are bad because the communication costs associated with the merging step become higher as the number of nodes increases imposing smaller speedup values.

The PC approach performed relatively well but the speedup exhibits low values that correspond to low efficiencies. On the other hand, the SLMF strategy, the ScanlineFlow Rasterization algorithm, provided excellent results: not only the speedup values increase as more nodes are added but good efficiencies are also achieved, as it can be observed in the figure 3. The results are due primarily to the fact that rendering phase executes concurrently with the pixel merging: the scanline synchronization overhead had a small influence for a moderate number of processors (<= 16).

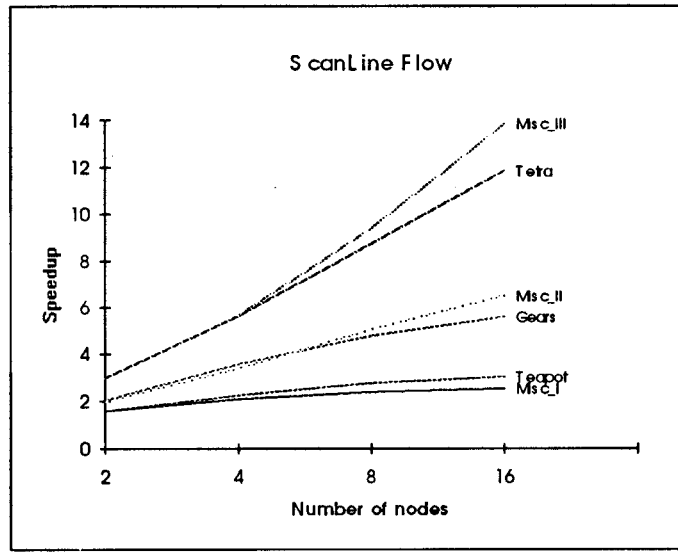

Figure 3 - Speedup values for Scanline Flow rasterization algorithm

It is interesting to refer that, at a low number of processors (2 and 4), the algorithm execution, for Tetra and Misc_III scenes, provided efficiencies greater than 100%. This seems to result from the fact that we have a lot of memory management associated to the algorithm execution: traversing the bucket-sorted list and maintaining an active polygon list are not linear operations.

The following figure illustrates how well the SLMF scheme behaves when compared with the two SLML solutions. It is important to refer that while sequential execution of the scanline z-buffer is slower than the sequential execution of the triangle rendering z-buffer algorithm, the corresponding parallel versions provided opposite results for a number of nodes greater or equal to 8.

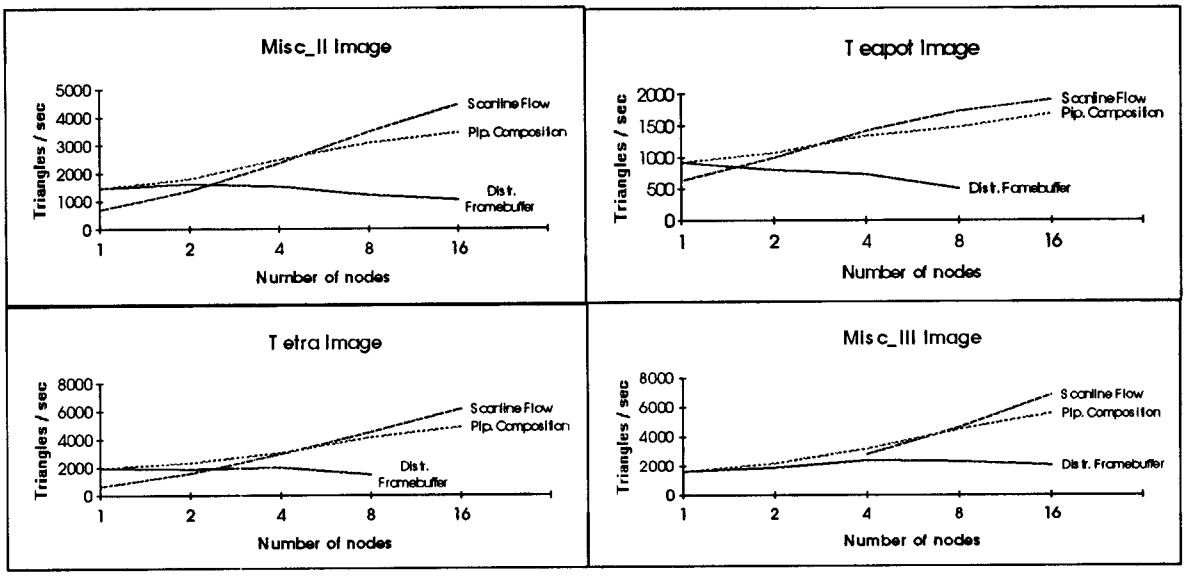

Figure 4 - Strategies comparison

6. Conclusions

For a general purpose distributed memory computer system, both full-frame strategies, SLML and SLMF - merging a full frame from each node is very regular and easy to implement - gave us very important indicators about research directions for the implementation of sort-last algorithms on multicomputers. One of these indicators concerns to the merging phase. The answer to the question if the rendering/merging phases are to run concurrently or consecutively was given in this paper: the SLMF-full strategy, through the ScanlineFlow Rasterization algorithm, has demonstrated to be capable to provide good results: not only the speedup values increase as more nodes are added but good efficiencies are also achieved. The SLML-full works relatively well for machines with small number of processors, but for larger multicomputers this method imposes severe performance restrictions because the execution of this operation takes a large percentage of the total execution time. PC algorithm performed much better than DF algorithm since the communications of the former solution are much lower than the later.

The other indicator regards to the load balancing problem. We saw that, in general, scattering provided good computational load balancing. An extra dynamic load balancing scheme will be only effective if the input scenes contain a few large polygons.

7. References

[Badouel92] - D. Badouel, C. Wuthrich, E. Fiume, "*Routing Strategies and Network Contention on Low-Dimensional Interconnection Networks*", CSRI - TR 258, Univ. Toronto, 1992.

[Cox92] - M. Cox and Pat Hanrahan, "Depth Complexity in Object-Parallel Graphics Architectures", *Proc. Seventh Workshop on Graphics Hardware,* Eurographics Technical Report Series, 1992, pp. 204-222.

[Cox93] - M. Cox and Pat Hanrahan, "Pixel Merging for Object-Parallel Rendering: A Distributed Snooping Algorithm", *Proc. Parallel Rendering Symp.,* ACM Press, New York, 1993, pp. 49-56.

[Cox95] - Michael B. Cox, "*Algorithms for Parallel Rendering",* doctoral dissertation, Princeton University, May 1995

[Dally87], W. Dally and C. Seitz, "Deadlock Free Message Routing in Multiprocessor Interconnection Networks", *IEEE Transaction on Computers,* Vol. 36, nr. 5, May 1987, pp. 547-553

[Evans92, Fuji93, Kubota93] - Graphics Workstations Series Technical Notes

[Haines87] - Eric Haines, "A Proposal for Standard Graphics Environments", in *IEEE Computer Graphics and Applications,* Vol. 7, n° 11, November 1987, pp. 3-5.

[Molnar90] - S. Molnar and H. Fuchs, "Advanced Raster Graphics Architecture", in *Computer Graphics: Principles and Practice,* 2nd ed., J. D. Foley et al., eds., Addison-Wesley, Reading, Mass., 1990.

[Molnar91] - S. Molnar, *Image-Composition Architectures for Real-Time Image Generation,* doctoral dissertation, TR 91-046, University of North Carolina at Chapel Hill, Oct. 1991.

[Molnar92] - S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Computer Graphics* (Proc. Siggraph), Vol. 26, N° 2, July 1992, pp. 231-240

[Molnar94] - S. Molnar, M. Cox et al., "A Sorting Classification of Parallel Rendering", *IEEE CG&A,* Vol. 14, N° 2, July 1994, pp 23-32

[Pereira95] - J. Pereira, C. Wuthrich and M. Gomes, "Implementing Sort-Last Algorithms for Polygon Rendering on a Multicomputer", Technical Report RT12/95, May 95.

[Rogers85] - David F. Rogers, "Procedural Elements for Computer Graphics", *McGraw-Hill Int. Publications,* 1985

[Sutherland74] - I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, " A Characterization of Ten Hidden Surface Algorithms", *ACM Computing Surveys,* Vol. 6, N° 1, Mar. 1974, pp. 1-55.

Appendix A - Plates

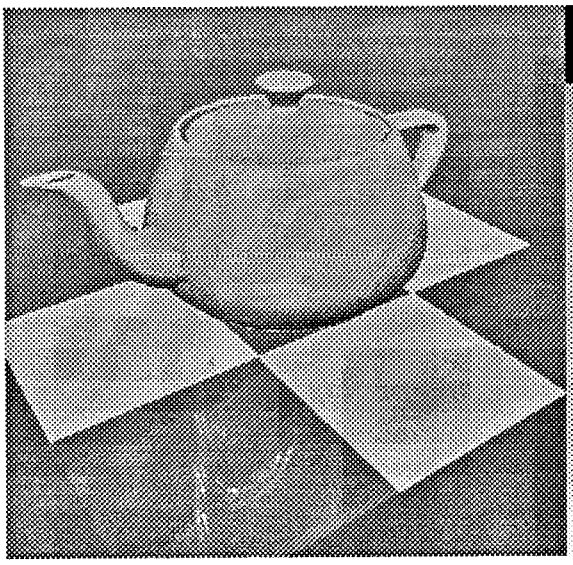

Plate A - Teapot image

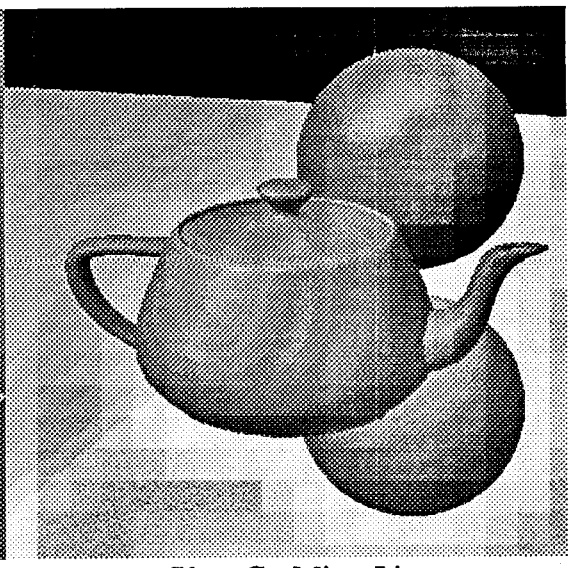

Plate C - Misc_I image

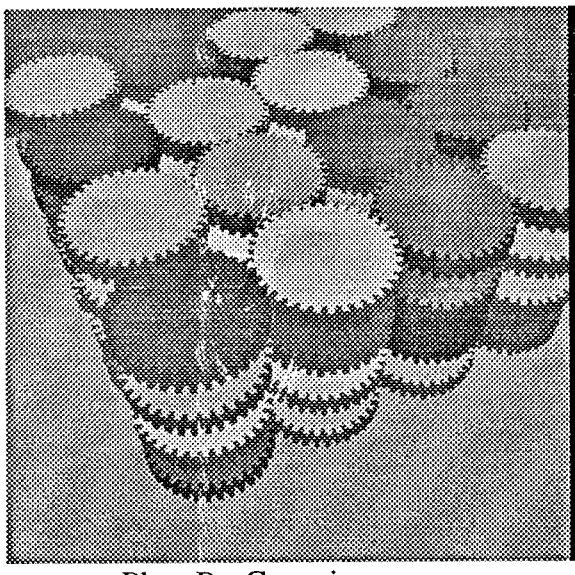

Plate B - Gears image

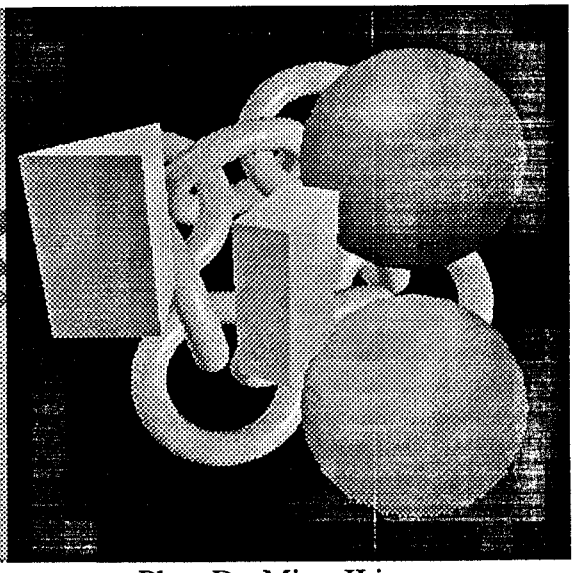

Plate D - Misc_II image

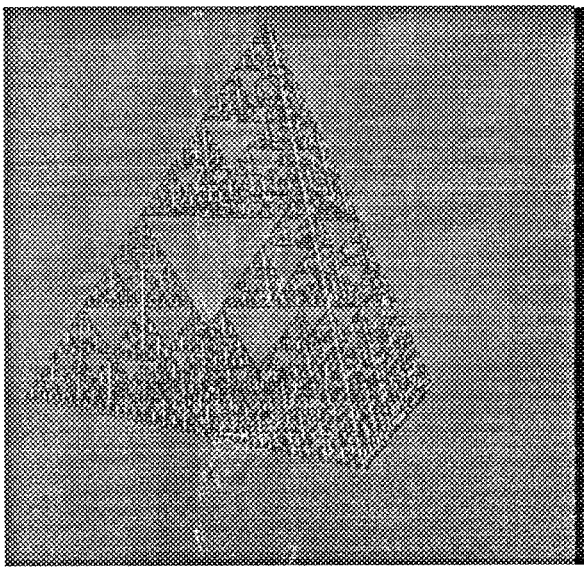

Plate E - Tetra image

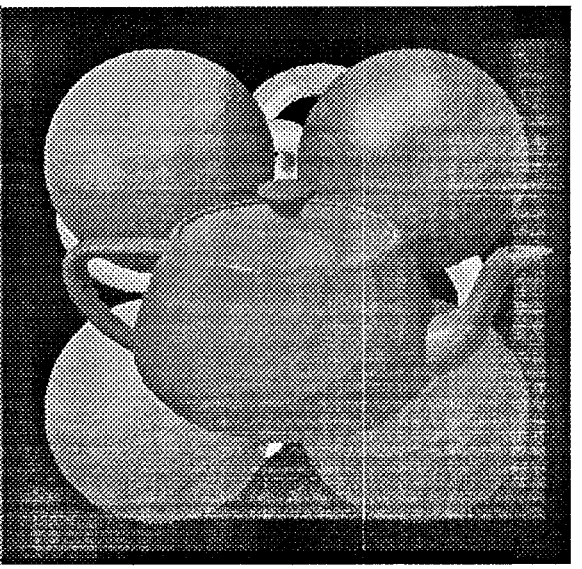

Plate F - Misc_III image

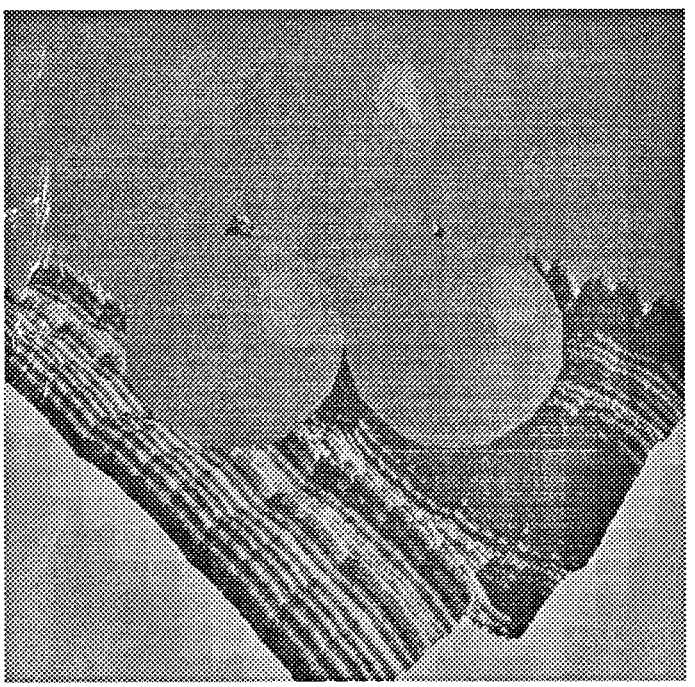

Plate G - Mountain image