Proceedings of International Conference
on Networking and Multimedia

Parallel Graphics Software Design

Tong-Yee Lee
Dept. of Information Management
Nantai Institute of Technology
Tainan County, Taiwan, R.O.C.
Email: tlee@candy.ntc.edu.tw

Abstract

Many grand challenge applications such as weather prediction are known for their extensive computations and large datasets, and are usually run on massively parallel processor systems (MPPs). To visualize data from these applications, potentially massive amounts of data must be transferred from the MPPs to special graphics workstations, often across a network. Under this condition, it may be preferable to visualize the data in place. In this paper, we first reveal the usefulness of directly perform rendering on parallel machines, and then discuss design issues related to this field. Finally, we present a sort-last class polygon rendering method which performs better than our previous work by 13-30 % times faster. To the best of our knowledge, the reported rendering rate obtained from a 512 node Intel Delta, is much higher than any other parallel implementation on the same kind or similar parallel architectures.

Keywords: parallel graphics, polygon rendering, parallel image composition

1. Limitations of Traditional Graphics Post-Processing

Specialized graphics hardware and workstations have been built to provide very impressive performance results for rendering images. However, use of parallel machines for rendering images provides many advantages over specialized graphics hardware [1]. Scientific simulation datasets are generated on large parallel computers and their sizes range from hundreds of megabytes to hundreds of gigabytes [2,3]. To render these datasets on graphics workstations, massive amounts of data must be transferred across communication networks. Limitations on I/O rates and network bandwidths force researchers to select portions of datasets, for example, by analyzing data to only certain time-steps. Therefore, unexpected or subtle phenomena can be potentially overlooked. By rendering images on parallel computer, massive data shipment is avoided and data can be analyzed in more detail to explore new phenomena. In addition, since rendering software can be integrated directly with the simulation software, there is no need for extra storage, and rendering can be a part of the simulation. By the combination of simulation and rendering together, it allows users to interactively "steer" simulation, rather than have to wait for long simulations and with high storage and transmission cost, only to find during post-processing (rendering at graphics workstations) that the simulations are wrong or uninteresting.

2. Parallel Rendering Is Useful

The aggregate computing power and memory capacity on modern parallel computers allow us to exploit many computationally expensive rendering schemes such as ray tracing and volume rendering to explore rendering large datasets. Even with the latest acceleration techniques running on the top-of-the-line workstations, it still takes a few seconds to a few minutes to render an image with these computationally expensive rendering schemes. However, with these rendering techniques, large

MPPs can render the images faster than workstations can, possibly, in real-time, or at least achieving interactive frame-rates [2,3]. In addition, with the advent of larger parallel machines and better scanners and instrumentation, larger and larger datasets are being generated today, some of which would not fit in memory of a workstation class machine. On the other hand, MPPs have larger memory capacities to render such larger datasets.
Software-based parallel rendering provides more flexibility in allowing additional geometric primitives, different rendering schemes, non-standard lighting model, etc. Usually, special graphics workstations perform well only on a small set of geometric primitives, use special rendering schemes and a limited set of illumination models. Therefore, parallel rendering is useful and there is growing interests in parallel rendering algorithm design.

3. Polygon Rendering(Z-Buffer) and Its Parallelization

In this paper, the rendering algorithm we parallelized is polygon rendering (Z-buffer). Polygon rendering is an image space rendering technique [4]. It provides faster rendering speed and acceptable photorealism. Polygon rendering produces a 2D representation of the 3D scene model taking into the account the lighting and perspective distortion. This scheme is popular and supported in most commercial graphics workstations. A standard polygon rendering (Z-buffer) pipeline is described as follows: The vertices of polygons are illuminated by different light sources, transformed from 3-D world space to 2-D screen space, and truncated by a clipping pyramid. The polygons are then scan converted to pixel values and a Z-buffer hidden surface elimination is performed.
In the past, many efforts were proposed to parallelize polygon rendering. Whitman's book [5] multiprocessor rendering methods and our recent paper [6] thoroughly surveyed the previous work in parallel polygon rendering work. We have discussed design issues in [7]. Here, several key issues are listed below:

- **Parallelization Algorithms**
- **Load Balancing**
- **Data Communication**

3.1 Parallelization Algorithms

How to parallelize a polygon rendering? There are many ways to do it. Molnar et al. [8] describe a framework for parallel rendering where the sort and redistribution of data occurs when transforming 3D objects to 2D screen space. They delineate three types of parallel rendering algorithms: sort-first, sort-middle (image-oriented) and sort-last (pixel-oriented). They conclude that there is no obvious evidence that some class is always better than others. In the sort-first algorithms (shown in Figure 1), each polygon is first pre-processed to determine which screen region it will be projected on. The primitive is then sent to the processor corresponding to this projected region, which performs all pipeline operations on that polygon. In the sort-middle parallel algorithm (shown in Figure 2), each polygon's geometry transformation is first done locally, then the algorithm determines where the transformed polygon will be sent. The sort-last class of algorithms (shown in Figure 3), in general, delay the data sort until the geometry processing and the rasterization of all polygons are completed. After each processor finishes rendering its allocated polygons, the sub-images created by the processors are merged into the final image.

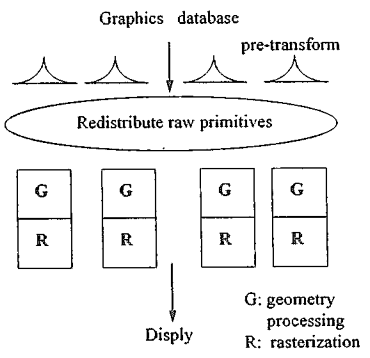

Figure 1. The sort-first parallel pipeline

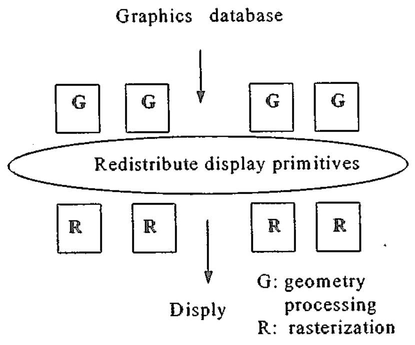

Figure 2. The sort-middle parallel pipeline

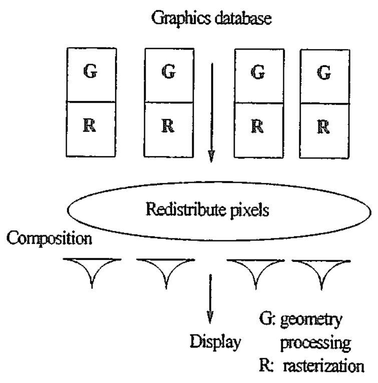

Figure 3. The sort-last parallel pipeline

3.2 Load Balancing

In Molnar's classification, the graphics primitives and screen space is distributed among the processors. For simplicity, here, we assume that they are evenly distributed among the processors and each primitive is a triangle. Each processor can perform both geometry transforming (shading, model and coordinate transforming, culling and clipping) and rasterization (rasterizing the triangles and hidden surface removal). For shading and transforming part, since the triangles have been distributed evenly to the processors, and these operations may be performed independently on each triangle, this part can be parallelized perfectly. Back-face culling and clipping can introduce local variations in workload which will detract from perfect speedup. Fortunately, this part does not contribute too much imbalances. Similar variations can be introduced in the rasterization and Z-buffer computations, and will likewise be the most significant contributors to slow down the speedup. The impact of these variations is both scene and view dependent. See an example in Figures 4 and 5. In Figure 4, the projected primitives are evenly distributed among the screen. However, in Figure 5, the projected primitives are concentrated on the center of the screen, and therefore, most rasterization workload is done by the processors which rasterize the central parts of the screen. In general, in graphics pipeline, the rasterization is the most timing contributor. Therefore, to achieve a high parallel efficiency, it is important to devise a load balancing to even out rasterization workload among the processors. In the past, there have been many work reported to balance the rasterization workload. The schemes were classified into: static [9], dynamic [10], and adaptive [11,12] classes. For more discussion, see [6].

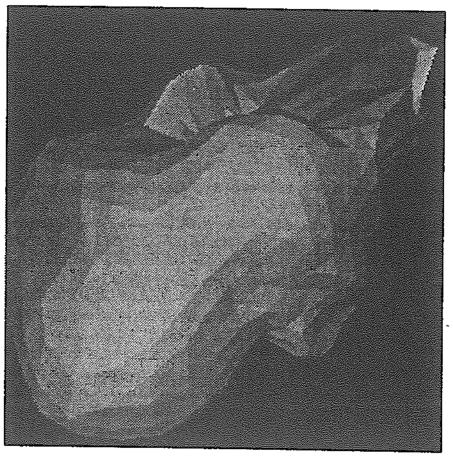

Figure 4. The primitives are evenly projected on the screen space

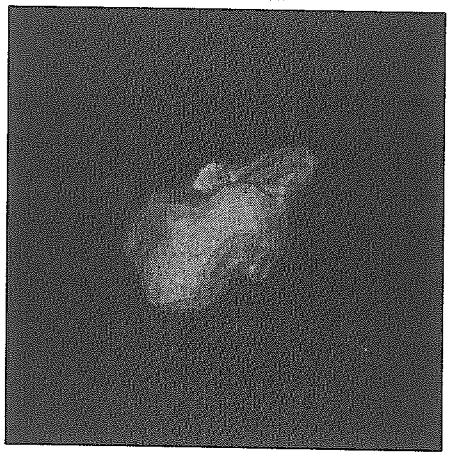

Figure 5. The primitives are clustered on the center of the screen.

3.3 Data Communication

In parallel rendering, the data (either graphics primitives or the screen space) is partitioned and distributed, so the data communication is introduced. The communication overhead consists of several parts: packing and unpacking data, sending data, waiting time, termination and something else. In general, if we assume that a processor will, on average, sends v data to each of p destinations, including itself. So, the total number of message is inserted to the whole system is vp^2. This is a kind of all-to-all communication. As the size of the system grows, this number grows even more quickly. Unfortunately, the communication still can not be solved in a scaleable manner even many advances in collective operations on parallel architectures [13]. So, this nonlinear cost does not decrease linearly with the numbers of processors, and therefore detract from the perfect speedup. For more details, please see [9]. In the past, we [16] and Ellsworth [10] have presented similar approaches to solve the communication problems. In both cases, the buffered data goes to an intermediate processor before being forwarded to its destination, and in both cases the communication complexity is reduced by not sending the data directly to its final destination. In both cases, the number of messages required is reduced approximately from $O(p^2)$ to $O(p^{3/2})$. In [14] and 15] , both proposed similar divide-and-conquer approaches to compose image pixels at optimal stages (*i.e*, ***log P***). Both schemes work well on flat-tree or hybercube like parallel architectures but not on the mesh parallel architecture [16].

4. Our Sort-Last Parallel Polygon Renderer

In this section, our parallel graphics system is introduced and was developed on the Intel Delta parallel computer at Caltech, a 2-D mesh message-passing architecture. It belongs to sort-last class rendering system in term of Molnar et al.'s classification [8]. For a sort-last class, first we evenly distribute graphics primitives (static load balancing scheme similar to [9]) to all processors, and partition the image screen into disjoint regions. As the rendering is done by each processor, a global image composition is performed to merge all subimages into the final display. Figure 6 shows an example of four processors rendering the Teapot scene in a sort-last manner.

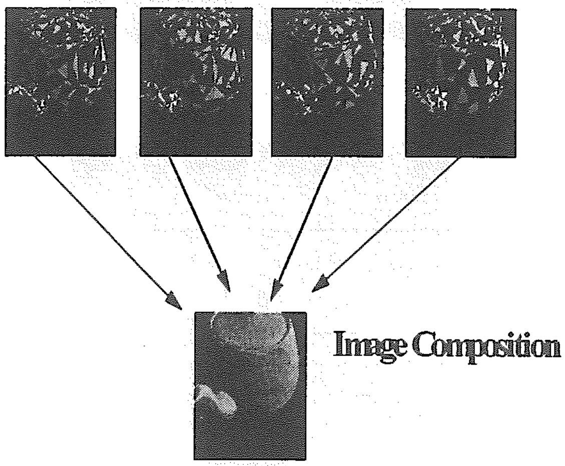

Figure 6. An example of our sort-last polygon renderer

In our previous work [16], we presented a scheme called DPF (Direct Pixel Forwarding) and its variant called DPFL (see [16] for more details). An example of DPF using 4 processors is shown in Figure 7. In Figure 7, at each stage *j*, each processor P_i sends active pixels in its $Z_{(i+j+1)mod4}$ to the processor $P_{(i+j)mod4}$. and receives a $Z'_{(i+1)mod4}$ from the processor $P_{(i-j)mod4}$. Then composes the local $Z_{(i+1)mod4}$ with the incoming $Z'_{(i+1)mod4}$

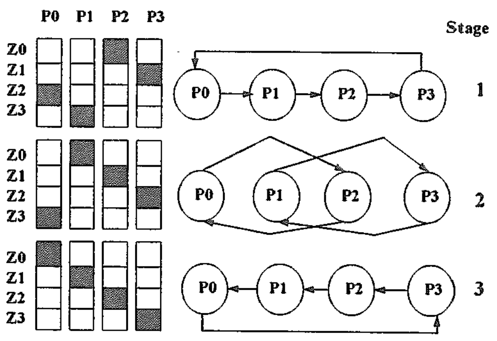

Figure 7. Direct pixel forwarding composition using 4 processors (1x4)

Here, we describe another approach termed as DPFS (DPF with Task Scheduling) with an attempt to achieve higher rendering rate. The idea of the DPFS scheme is to exploit more communication links than DPF and to reduce message sizes by overlapping both communication and rendering work. From our experience, this approach is very useful to reduce the communication time in the large system size. The DPFS scheme is described as follows:

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

```
        while (A-type messages remain to arrive from other processors){
            if message needed to be forwarded in the second phase of DPF
            then unpack this message into B-type buffers;
            else Z-buffer this message with local region;
        }
        flush all B-type buffers to other processors in the order of DPF's second phase;
        while (B-type messages remain to arrive from other processors){
            Z-buffer this message with local region;
        }
    synchronize; /* make sure all processors finish this frame */
```

For 2-D mesh parallel architectures like the Intel Delta, we logically group the 2-D mesh (*r x c*) into many sub-rings. There are *r* processors in row direction and *c* processors in column direction. Each row or column forms a subring. Under this arrangement, our previous work [16] (*i.e.*, DPF scheme) performs image composition in row direction in the first phase, and in column direction in the second phase. In the DPFS scheme, there are two types of message buffers which consist of (*r-1*) *A*-type message buffers and (*c-1*) *B*-type message buffers. The *A*-type message buffers store both pixel values and (*x,y*) coordinates of corresponding regions in the first phase of the DPF scheme, and the *B*-type message buffers are in charge of the second phase. In the first *while loop*, the rendered pixels of each local triangle can be temporarily stored either in the *A*-type buffers (*i.e.*, these pixels do not belong to local processor's region in the first phase of the DPF) or in the *B*-type buffers (*i.e.*, these pixels do not belong to local processor's region in the second phase of the DPF) or can be Z-buffered in local processor's assigned portion of the final image. We implement this scheme by asynchronous routines for message send and receive, and these can be used to overlap message transfers with both triangle rendering and pixel merging computations. Unlike the DPF scheme, groups of pixel message are sent asynchronously and are not delayed until the end; therefore, shorter messages are needed to be flushed (*i.e.*, network congestion can be less). For more detailed discussion, please see [6]. Similarly, we also implement a variant of DPFS called DPFSL. The DPFSL requires interleaving the scanlines in the first phase.

5. Experimental Results

To perform our experiments, we used datasets from Eric Haines's SPD database [17]. The SPD database has been used in many previous studies and is believed to be a good representation of real data. Table 1 shows the sizes of two datasets among our test scenes. Figures 8 and 9 show the rendering results for the "Teapot" and "Tree" scenes. In our implementation, each large group consists of 2000 triangles and the data for each triangle is 48 bytes.

Scene	Number of Triangles	Size of Dataset (MBytes)
Tree	426K	19.5
Teapot	160K	7.3

Table 1. Number of triangles and data size of the two test scenes.

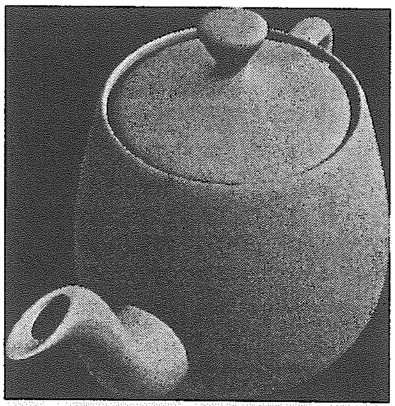

Figure 8. Teapot and Tree scenes

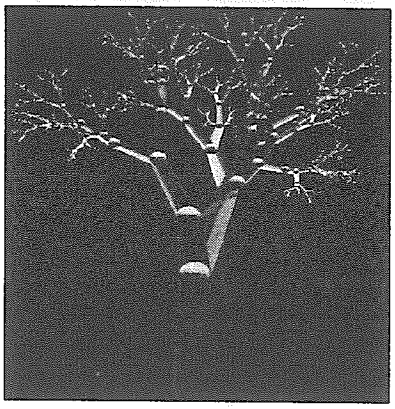

Figure 9. Teapot and Tree scenes

For page limit, we only present part of our experimental results here. Table 2 shows that DPFSL consistently performs better than DPFL for large systems. DPFSL achieves a rendering rate of 3.8 to 4.5 million triangles/sec using 512 processors. In comparison with DPFL, we gain about one half to one million triangles/sec rendering rate. For example, replacing DPFL with DPFSL, the performance of rendering ``Tree" scene changes from 3.9 to 4.5 million triangles/sec. Our rendering rates do not drop off significantly for up to 512 processor. In contrast, most previous work declined their performance significantly after small number of processors in use. On the other hand, for small systems, the DPFSL is slightly slower, by at most 5%, for ``Teapot" scene using 8 processors. This is due to the extra overheads incurred in three while loops. These include buffer management, message detection, breaking of rendering pipeline by inserting message handling code in the first while loop and so on. In the case of small systems, the saving of message communication time can not offset these factors and results in slight slow-down in performance.

#Procs	8	16	32	64	128	256	512
Teapot							
DPFL	115K	223K	4156K	756K	1287K	1970K	2705K
DPFSL	109K	213K	404K	771K	1438K	2574K	3893K
Tree							
DPFL	--------	--------	555K	1026K	1797K	2898K	3871K
DPFSL	139K	276K	528K	1002K	1804K	3131K	4482K

Table 2. The comparison in rendering rate between DPFL and DPFSL for two test scenes

Figure 10 shows the relative speedup plot for rendering both scenes using DPFL and DPFSL schemes. The relative speedup values are based on the times obtained from the minimum configuration that test scene can fit in. For example, it needs 8 processors to hold both scenes in DPFSL implementation. For both test scenes, the DPFSL rendering rate scales better than the DPFL. For example, the DPFSL scheme scales well first with about 98% speedup efficiency, but decreases to about 50% (Tree scene) at the largest configuration of the Delta. For further discussion of this figure (take DPFSL scheme as an example), we divide the total rendering time into two main parts: rendering time (Render) that consists of exact rendering time and pre-processing time for composition, and the composition (Compose) time. Figure 11 and 12 show the time breakdown of our renderer for the ``Teapot" and ``Tree" scenes using DPFSL scheme. For rendering both scenes, most of time is spent rendering (*i.e.*, geometry transform and pixel rasterization); the composition time (*i.e.*, merging and communication) are consistently small. From both figures, we see that the rendering time decreases slightly linearly as the number of processors are increased for both scenes. However, the composition time slows down slightly. As the size of the machine increases, one of the reasons for decreasing speedup efficiency is that rendering load becomes uneven among the processors when fewer triangles are computed on each processor. The major cause is that composition time does not scale with increasing number of processors. With the increase in the number of processors, the composition time of each scene is almost kept constant while the rendering time decreases, and when composition time becomes dominant or comparable to rendering time, the speedup will decrease. For more details, please see [6] and [7].

6. Conclusion

In this paper, first, we discussed the motivation of our work and then outlined the key issues in this area. This paper presented a implementation of parallel graphics system on the Intel Delta supercomputer. We described a scheme that improved our previous work and obtained higher polygon rendering rate. Using 512 processors of the Delta, we obtained significant performance improvement ranging from 13% to 30% on SPD test scenes. However, there is a slight performance degradation when using small number of processors. This is due to extra overheads incurred in three while loops. Therefore, to achieve better performance, we need to make a right choice among DPFL and DPFSL. In our future work a performance model will be developed and analyzed to assist in choosing a right scheme to get the best possible rendering rate. In addition, we may investigate hybrid scheme (combine sort-last with sort-middle) or rendering directly at compressed Z-buffered data to reduce the amount of communication.

Acknowledgments

This research was performed in part using the Intel Touchstone Delta System operated by California Institute of Technology on behalf of the Concurrent Supercomputing Consortium.
Access to this facility was provided by Pacific Northwest Laboratory (PNL), a multi-program laboratory operated for the U.S. Department of Energy by Battelle Memorial Institute under

Contract DE-AC06-76RLO 1830. This project is mainly sponsored by the Boeing Chair Professor fund, and in part, by the National Science Council, project No. NSC-86-2213-E-218-011, ROC, Taiwan.

References

1. Crockett, ``Design Considerations for Parallel Graphics Libraries," *the Proc. Intel Supercomputer Users Group*, pp.3-14, June 1994.

2. *The Proceedings of 1993 Parallel Rendering Symposium,* IEEE press, San Jose, CA, October 1993.

3. *The Proceedings of 1995 Parallel Rendering Symposium*, IEEE press, Atlanta, Georgia, October 1995.

4. J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, `` *Computer Graphics: Principles and Practice*," 2nd Ed., Addison-Wesley 1990.

5. S. Whitman, ``*Multiprocessor Methods for Computer Graphics Rendering*," Jones and Bartlett, 1992.

6. Tong-Yee Lee, C.S Raghavendra, J.B. Nicholas, ``Sort-Last Polygon Rendering on 2D Mesh Parallel Computers," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 2, No. 3, pp. 202-217, September 1996.

7.Tong-Yee Lee, "The Parallel Polygon Rendering System," *submitted to Parallel Computing*.

8. S. Molnar, M. Cox, D. Ellsworth and H. Fuchs, ``A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, pp. 23-32, July 1994.

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

10. D. Ellsworth, ``A New Algorithm for Interactive Graphics on Multicomputers," *IEEE Computer Graphics and Applications*, July 1994.

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

12. S. Whitman," Dynamic Load Balancing for Parallel Polygon Rendering," *IEEE Computer Graphics and Applications*. Vol. 14, No. 4, July 1994, pp. 41-48.

13. M. Barnett, R. LittleField, D.G. Payne and R. van de Geijn, "Global Combine on Mesh Architectures with Wormhole Routing," *Proc. of 7th International Parallel Processing Symposium*, 1993.

14. K. Ma, J. S. Painter, C.D. Hansen and M. F. Krogh, " Parallel Volume Rendering Using Binary Swap Composium," *IEEE Computer Graphics and Applications*, July 1994, pp. 59-67.

15. R. J. Karia, " Load Balancing of Parallel Volume Rendering with Scattered Decomposition," *Proc. Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

16. Tong-Yee Lee, C.S. Raghavendra, and John B. Nicholas, ``Image Composition Methods for Sort-Last Polygon Rendering on 2-D Mesh Architectures," *Proc. of the 1995 Parallel Rendering Symposium*, ACM, pp. 55-62, Oct. 1995.

17. E. Haines, "A Proposal for Standard Graphics Environment," *IEEE Computer Graphics and Applications*, 7(11): 3-5, Nov. 1987.

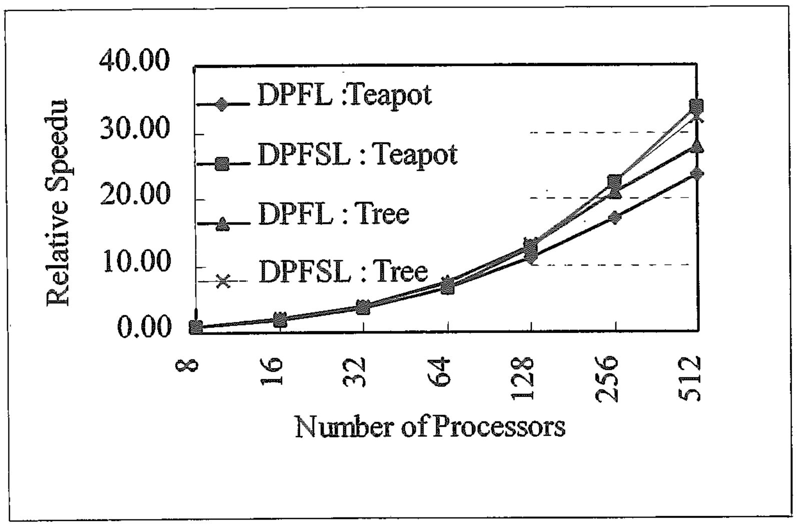

Figure 10. Relative Speedup

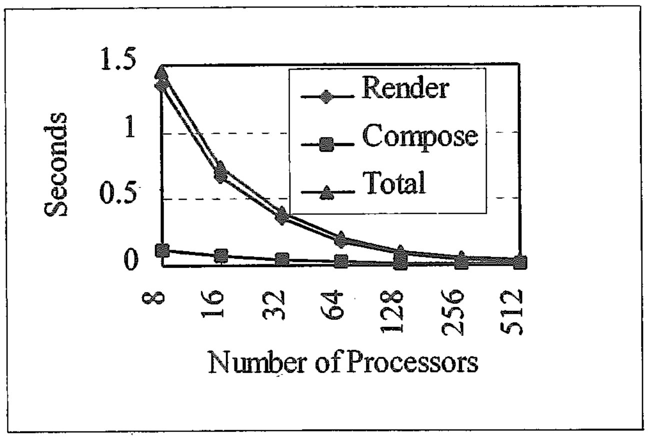

Figure 11 Time breakdown for the Teapot Scene

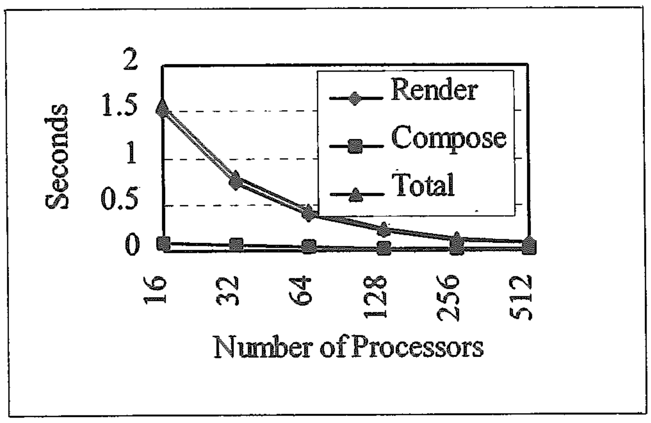

Figure 12. Time breakdown for the Tree Scene