Implementation and Evaluation of the Parallel Mesa Library

Tulika Mitra and Tzi-cker Chiueh
Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
mitra,chiueh@cs.sunysb.edu

Abstract

This paper describes the implementation and performance evaluation of a 3D graphics library that can be readily linked with parallel applications to provide run-time visualization on large-scale message-passing parallel machines, such as Intel Paragon. The prototype implementation is currently fully operational, and is based on a public-domain OpenGL implementation Mesa, and a sort-last parallelization strategy. Through a detailed performance analysis, we show that the scalability of the current prototype is close to the theoretical limit for the given hardware architecture. We also developed a unified framework to describe parallel compositing algorithms and show that two popular parallel compositing algorithms, binary swapping and parallel pipeline compositing, are just two extreme instances of this framework. Such a framework is important because it allows users to tailor the compositing algorithm according to the computation/communication characteristics of specific parallel machines by tuning the parameters appropriately. The current parallel Mesa library prototype implements such a parameterizable family of compositing algorithms.

1. Introduction

3D polygon rendering consists of two distinct stages: *geometric transformation* and *rasterization*. Geometric transformation converts the objects in the input model from the world coordinate to the viewing coordinate, while rasterization scan-converts the transformed geometric primitives into pixel color and depth values to form the final image. The total polygon rendering time can be significantly reduced by exploiting the parallelism of the rendering pipeline. The parallelization strategies for polygon rendering can be classified as *sort-first*, *sort-middle* and *sort-last* [11].

In the *sort-first* strategy [2, 14, 15], the 3D primitives in the world coordinate space are partitioned depending on the image region to which they will get projected. Every 3D primitive is then distributed to the processor that is responsible for that image region, transformed and rasterized. Finally the image regions are combined to form the final rendered image.

In the *sort-middle* strategy [6, 5, 13, 1, 7] the 3D primitives are first transformed and then distributed to different processors based on the image region to which the transformed coordinates of the primitives belong and rasterized to create a sub-image. The sub-images are then combined to form the final image.

The *sort-last* [12] strategy partitions the 3D input model in the beginning of the rendering pipeline without taking into account the viewpoint or object coordinates, performs geometric transformation and rasterization on each partition independently to produce a partial image, and finally composites the partial images according to the depth value of each image pixel. The compositing stage plays an important role in the effectiveness of the *sort-last* strategy.The simplest method is to send the sub-images to a single compositing processor [12] or to do binary tree composition [16], where more and more processors become idle as the composition progresses. Ma et. al. [10] and Karia [8] described similar parallel compositing techniques for volume rendering applications that can be easily adapted for polygon rendering, and are now referred to as binary swapping. Lee et. al. [9] showed that parallel pipeline compositing is more efficient that binary swapping in parallel machines with 2D mesh networks such as Intel Delta because it reduces the communication conflicts.

In this paper, we describe our 3D parallel polygon rendering library that uses the *sort-last* strategy. We show that binary swapping and parallel pipeline compositing are just two extremes of the same parallel compositing algorithm family, and the optimal design point may well be somewhere in the middle of the spectrum, depending on the communication network architecture and the 3D data set characteristics. The rest of this paper is organized as follows.

Section 2 reviews previous works on parallel 3D graphics rendering. Section 3 describes the design and implementation of the parallel graphics library. In section 4, we present a general framework for parallel compositing algorithms. In Section 5, we show the results of a detailed performance evaluation of the prototype implementation. Section 6 concludes this paper with the main research results.

2. Parallel Mesa Library

2.1. Overview

The goal of this work was to implement a parallel version of the Mesa library, a software-only 3D graphics rendering library with an API (Application Programming Interface) similar to OpenGL, which is the de facto 3D graphics API standard on UNIX-based workstations. We chose Mesa because its source code is freely available for modification. A parallel 3D graphics library that can be linked with computation programs can greatly simplify the programming effort required to render the intermediate results of large-scale computation problems on such multicomputers as Intel Paragon. Numerical simulation programs running on parallel computers can directly call the parallel Mesa library, which then provides the end users immediate visualization feedbacks of the intermediate results. With this immediate feedback, users may choose to change some of the program parameters and thus interactively "steer" the computation towards a direction that is more likely to produce the targeted results. Specifically, the advantages of this approach are the following: (1) The data transfer overhead is reduced because both rendering and computation are performed on the same parallel machine; (2)The OpenGL interface provided by the parallel Mesa library isolates the computation programs from the underlying graphics renderer implementations on different parallel machines; (3)Because calls to graphics library are embedded within the computation itself, interactive visual feedback of the computation result is feasible.

In contrast, earlier parallel 3D graphics systems require the computation results to be shipped to a dedicated graphics subsystem, which then renders the final images off line. Interactivity is inherently impossible in this setup. PGL [4] on the other hand, is a parallel graphics library using sort-middle architecture. However, the API for PGL is not a standard. In contrast, we kept Mesa (OpenGL) API unchanged in our library, while we added just two more functions. This way it could be readily used by OpenGL programmers.

The *sort-last* parallel rendering strategy fits very well in this library framework. Each node participating in the numerical computation makes appropriate Mesa (OpenGL) calls to render the intermediate computation results. After sending all the rendering commands, each node calls a parallel Mesa library function to indicate that it is ready to enter the compositing stage. When all the nodes have called this function, the parallel Mesa library starts the compositing of all the sub-images. In this architecture, rendering and numerical computation are temporally interleaved, while compositing is performed in a batch fashion.

The template of a parallel program that uses the parallel Mesa library looks like the following:

```
#include <pmesa.h>
void main()
{
        InitMesa(width, height);
                :
             Computation
                :
             Mesa calls
                :
           DisplayMesa();
}
```

`pmesa.h` is the include file associated with the parallel Mesa library. At the beginning, the program should call `InitMesa(width, height)` to initialize the data structure. This call is needed only once in the program. Then the program performs computation and makes Mesa calls as new results become available. When the program completes one step of the computation and is ready to render the entire intermediate result, it calls `DisplayMesa()`. After all the processor nodes have called `DisplayMesa()`, the parallel Mesa library merges all the sub-images. Using the parallel Mesa library involves nothing but the linking of the computation program with the library at compile time.

2.2. Design Issues

The actual implementation of the sort-last method is relatively straightforward because it does not need to modify the Mesa internals. The only new codes that need to be developed are data distribution at the beginning of the graphics pipeline and compositing of sub-images at the end. However, there were a number of design issues involved in this implementation, which we will discuss in this subsection.

First, rendering and rasterization can be performed on the same set of processors (the integrated strategy), or on disjoint sets of processors (the disjoint strategy). The advantage of the disjoint strategy is that it is amenable to a pipeline implementation in which rendering and compositing are separately optimized so that the overall system throughput is maximized as required in the case of dedicated 3D graphics systems.

The integrated strategy, on the other hand, is advantageous when the rendering latency is the performance metric, because it puts all the computation power on both rendering and compositing. For interactive rendering of intermediate computation results, this choice is preferred. In fact, from the practical implementation standpoint, this choice is the only possibility because the rendering pipeline can not be arbitrarily split among the processors at run time when the rendering code is implemented as a library that is statically linked with computation applications.

The second design issue is the desirability of data redistribution when the optimal data distribution for the numerical computation that generates data outputs for visualization is different from the one associated with parallel Mesa. The performance tradeoff is the data re-distribution overhead versus the resulting saving in the rendering and compositing time. Because of the lack of experience with the library, we choose not to re-distribute the data set at this point for implementation simplicity.

The final and the most important issue is the parallel compositing algorithm. Instead of implementing any specific algorithm, like parallel pipeline or binary swapping, we developed and used a parameterizable family of algorithms for our library. This is a general framework that subsumes most of the parallel compositing algorithms. It helps user to choose a particular algorithm according to the communication characteristics and topology of a specific parallel machine, by just setting the parameters during compilation of the library for that machine. We describe this general framework in the next section.

3. A Framework for Parallel Compositing Algorithms

The compositing algorithms for the sort-last parallel rendering method can be taxonomized along two dimensions: (1) How rendered images are combined among compositing processors; (2) How rendered images are encoded during transfer. For the second dimension, there are two design alternatives: *full compositing*, in which the entire sub-images are transferred across the network for compositing, and *sparse compositing* [3], where only the portions of sub-images that contain "active" or non-background pixels are transferred. The performance tradeoff between the two schemes is the compositing overhead in terms of CPU time and network traffic, and the additional effort to "compress" rendered sub-images. We choose to implement the full compositing strategy because it is simpler, and also because our prototype experience with the sparse strategy indicates that its performance gain is not as significant because of the additional compression/decompression overhead.

For the first dimension, we have developed a general framework that can describe a large class of parallel compositing algorithms by specifying the tradeoff between the number of messages exchanged and the communication locality. We will start with the description of the building block of this general algorithm framework, the parallel pipeline compositing algorithm [9], and then proceed with the framework itself.

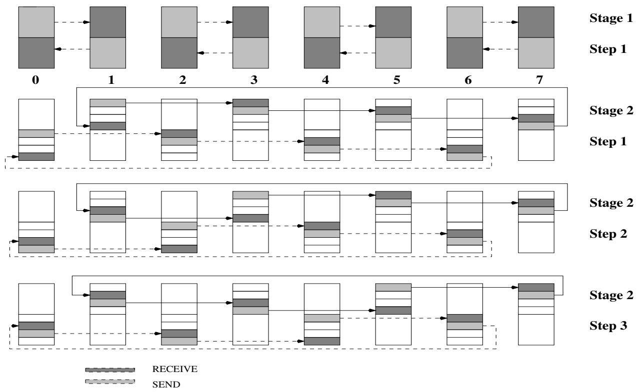

Figure 1. *The general parallel compositing algorithm for the 8-processor case with a 2x4 configuration.*

Parallel pipeline compositing divides the sub-images, including the depth and alpha values, generated at each of the M processors into M portions, Z_0, Z_1,... Z_{M-1}. Processors organized as a circular ring are denoted as P_0, P_1,... P_{M-1}. In the following algorithm, P_{next} of P_i is $P_{(i+1)\ mod\ M}$ and P_{prev} of P_i is $P_{(i-1)\ mod\ M}$.

```
for all processors do in parallel
        set current compositing area C_current as Z_i in P_i.
            for j=1 to M-1 do
                send C_current to its P_next
                receive the C_current of P_prev in C_temp
                set k = (i-j) mod M
                combine its Z_k with C_temp.
                set C_current to be the composited result i.e. Z_k
            end for
end for
```

All processors are busy at each stage. After $M-1$ steps, each processor holds $\frac{1}{M}$ of the final composited image. An important advantage of this approach is the small communication buffer requirement. It needs $\frac{1}{M} * 8A$ MB of communication buffer space per processor (where A is the number of pixels in the image and each pixel requires 8 bytes for depth and color values), which is a considerable saving compared to binary tree compositing that requires $8A$

MB of communication buffer space. Figure 2 illustrates this method for the 8-processor case.

Despite the fact that the parallel pipeline algorithm keeps all processors at high utilization efficiency, its performance is not necessarily optimal because it takes $O(N)$ steps to complete with N processors. The general framework described below subsumes the parallel pipeline algorithm by allowing faster propagation of compositing results. Let us assume that there are N compositing processors, where N is a power of 2. If it is not, we can always perform a folding at the first stage to have a power of 2 processors.

Given N, we first identify all the divisors of N. Next, an ordered subset of the divisors of N (except 1) are chosen such that their product is N. For example when $N = 16$, we can have 16, 8x2, 2x8, 4x4, 4x2x2, 2x4x2, 2x2x4 and 2x2x2x2. Assume N is factored into n_1 x n_2 x n_3, the corresponding parallel compositing algorithm proceeds as follows:

In the beginning, each of the participating processors holds a rendered sub-image, and they are numbered from 0 to $N-1$. The first step is to classify all processors between i to $i+n_1-1$ to one group, where $i = 0, n_1, \ldots, N-n_1$. Within each group, a *parallel pipeline compositing* is performed on the group's rendered images. At the end of the first stage, each processor now contains $\frac{1}{n_1}$ of the composited image for its group. At the second step, processors with IDs i, $i+n_1$, $\ldots$, $i+(n_2-1)*n_1$ form a group, where $i = j, j+1, \ldots, j+n_1-1$, and $j = 0, n_1*n_2, 2*n_1*n_2, \ldots, (n_3-1)*n_1*n_2$. Again within each group, a *parallel pipeline compositing* is performed on the group's partially composited image regions. At the third and last step, processors with IDs i, $i+n_1*n_2$, $\ldots$, $i+(n_3-1)*n_1*n_2$ form a group, where $i = 0, 1, 2, \ldots, n_1*n_2-1$. Once again within each group, a *parallel pipeline compositing* is performed. At the end of these three steps, each processor holds $\frac{1}{N}$ of the final composited image. A final collection stage is required to put all sub-images into one complete composited image for display. Figure 1 illustrates this method for the 8-processor case with a 2x4 configuration.

Given N processors, the *parallel pipeline compositing* corresponds to a N configuration of the general compositing algorithm, which is shown in Figure 2 and takes $(N-1)*T_{\frac{A}{N}}$ time, where A is the number of pixels in the image and $T_{\frac{A}{N}}$ is the time needed to combine two images each with $\frac{A}{N}$ pixels. The other extreme configuration, 2x2x..x2, corresponds to the *binary swap compositing* algorithm, which is shown in Figure 3 and takes $T_{\frac{A}{2}} + T_{\frac{A}{4}} + \ldots + T_{\frac{A}{N}}$ time. For a n_1 x n_2 x n_3 configuration, the time needed is $(n_1-1)*T_{\frac{A}{n_1}} + (n_2-1)*T_{\frac{A}{n_1*n_2}} + (n_3-1)*T_{\frac{A}{n_1*n_2*n_3}}$.

A variation of the binary swap scheme is the *binary-tree compositing* algorithm, which uses only one rather than two processors in each two-node compositing operation, and therefore takes approximately $logN * T_A$ time. One advantage of binary-tree compositing is that at the end of the algorithm, one node holds the final composited image.

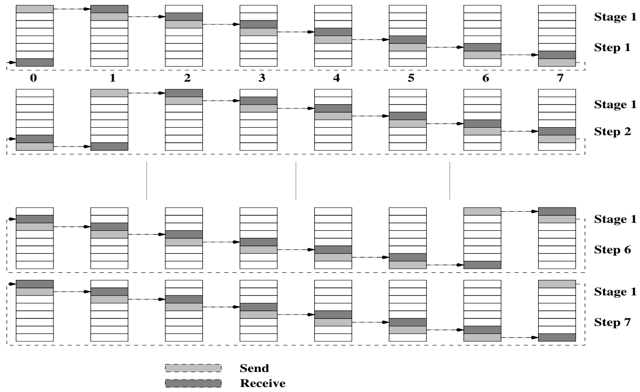

Figure 2. *The Parallel Pipeline compositing algorithm corresponds to a 8 configuration for the 8-processor case in the general parallel compositing algorithm framework.*

It can be shown that for a fixed number of processors the number of computation operations and the total number of bytes exchanged among processors associated with different configurations are identical. The performance tradeoff among different configurations thus lies in the communication efficiency, which in turn is determined by the number of messages communicated and the time taken by each message. For parallel machines with large transmission bandwidth among all pairs of processor nodes, such as Intel Paragon, reducing the number of messages is more important because it reduces the relatively large per-message overheads. The results in Table 1, which shows the compositing time for different configurations under different number of processors, demonstrates that this is indeed the case, i.e., the binary swapping algorithm always win. The reported time is the average taken over 10 runs for each configuration.

On the other hand, most of the message communications in the parallel pipeline algorithm are between neighboring nodes, as compared to the other extreme, binary swapping, whose communication pattern is less regular. For parallel machines whose communication network provides special optimization path for nearest-neighbor communications, better communication "locality" in the parallel pipeline algorithm may out-weigh the performance overhead due to higher message traffic. This has been demonstrated in [9], which showed parallel pipeline is better than binary swapping on an Intel Delta machine, where cross-section transmission bandwidth is much smaller and the message latency difference between local and global com-

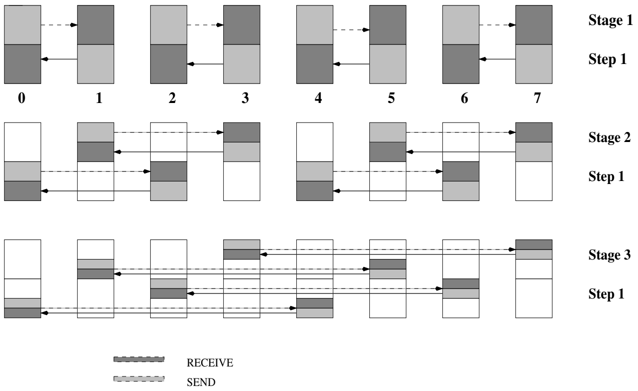

Figure 3. *The Binary Swap compositing algorithm corresponds to a 2x2x2 configuration for the 8-processor case in the general parallel compositing algorithm framework.*

munication is significant.

4. Performance Evaluation

4.1. Overall Performance

The current parallel Mesa library prototype is implemented on the 1,824-node Intel Paragon machine in Sandia National Laboratory using the MPI message passing interface. The data sets that we have tried with the parallel Mesa library fall into two categories: Three of them are scientific data sets (marked as **S**) created from a mesh generator out of volume data, and two of them are VRML (Virtual Reality Modeling Language) scenes (marked as **V**). The major difference between these two types of data sets is that most of the triangles in the scientific data set are smaller than 5 pixels, whereas the triangles in the VRML data sets have a much wider distribution in size. To set the baseline case, we measure the time to render each of the data sets on a single Paragon node. Table 2 shows these measurements.

To evaluate the performance of our parallel library, we wrote a user program that reads in 3D datasets (triangles and triangle strips) from disk, performs data distribution, and makes calls to our parallel library to render and display the image.

Table 3 and Figure 4 show the performance of parallel Mesa in terms of total rendering time and the triangle rendering rate to generate the final images with varying number of processors. With increasing number of processors, the total rendering time is decreasing and the rendering rate is increasing. The rate of performance change correlates very well with the processor number changing rate, i.e.,

No. of nodes	Configuration	No. of steps	Time (sec)
2	2	1	0.15
4	4	3	0.28
4	2x2	2	0.26
8	8	7	0.34
8	4x2	4	0.32
8	2x4	4	0.32
8	2x2x2	3	0.31
16	16	15	0.37
16	8x2	8	0.37
16	2x8	8	0.35
16	4x4	6	0.35
16	4x2x2	5	0.34
16	2x4x2	5	0.33
16	2x2x4	5	0.32
16	2x2x2x2	4	0.31
32	32	31	0.39
32	16x2	16	0.38
32	2x16	16	0.36
32	8x4	10	0.37
32	4x8	10	0.36
32	4x4x2	7	0.35
32	4x2x4	7	0.35
32	2x4x4	7	0.35
32	4x2x2x2	6	0.35
32	2x4x2x2	6	0.34
32	2x2x4x2	6	0.33
32	2x2x2x4	6	0.34
32	2x2x2x2x2	5	0.32

Table 1. *Compositing times for different configurations under different number of processors.*

exponential, for all data sets. However, beyond 64 nodes, the performance improvement rate starts to saturate for data sets that have fewer triangles. For larger datasets, the performance improvement due to parallelization is still quite significant even for 256-node configuration. In particular, it is possible to achieve more than 1 million triangles per second end-to-end performance with 256 nodes.

4.2. Detailed Analysis

The total rendering time of the sort-last parallel rendering method consists of three components: the transformation/rasterization time, the compositing time, and the time for sub-image collection, which puts together the final image. As the number of processors increases, the transforma-

Dataset Size	Number of Processors							
	2	4	8	16	32	64	128	256
69K (S)	5.35	2.89	1.67	0.98	0.67	0.54	0.45	0.42
125K(S)	7.41	3.99	2.22	1.27	0.81	0.60	0.49	0.44
470K(S)	25.50	12.99	6.83	3.70	2.05	1.24	0.81	0.61
398K(V)		11.94	5.29	2.96	1.43	0.90	0.63	0.51
964K(V)			16.22	8.82	3.98	2.16	1.27	0.85

Table 3. *Total rendering time in sec for five 3D data sets using different number of processors.*

Dataset Size	Rendering Time(sec)	Triangles/sec
69K (S)	10.30	6.70K
125K(S)	14.69	8.51K
470K(S)	50.83	9.25K
398K(V)	35.67	11.16K
964K(V)	112.43	8.57K

Table 2. *The rendering time on a single Paragon node for the five 3D data sets. S for scientific and V for VRML data sets.*

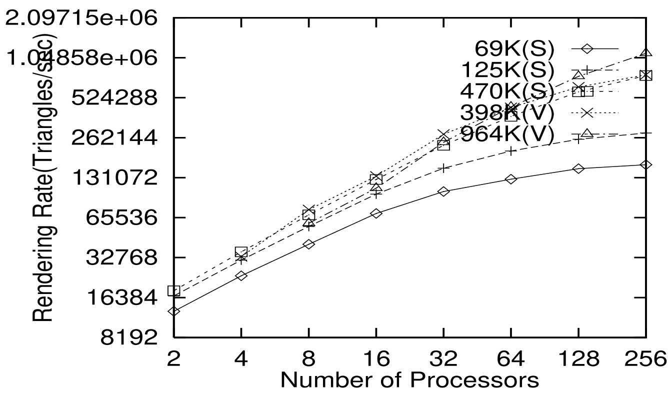

Figure 4. *The rendering performance in triangles per second for five 3D data sets using different number of processors. The axis for number of nodes is in logarithmic scale.*

tion/rasterization time decreases almost linearly, as shown in Figure 5, for both scientific and VRML data sets. Because we do not perform any load balancing during transformation/rasterization, the results in Figure 5 already fall on the conservative side. Even so, linear scalability is observed.

Although more processors decrease the transformation/rasterization time, they tend to increase the compositing and collection times, because more processors imply that compositing involves more images. With perfect load balancing, one can show the amount of compositing work per node increases as the number of processors used in transformation/rasterization increases. Let the number of pixels in the images to be composited be A. Then the number of depth buffer comparison when compositing two images is A. With perfect load sharing, the amount of work assigned to each of the two processors is $\frac{A}{2}$. In general, the amount of work per processor for n-image compositing among n processor is $\frac{n-1}{n} * A$. Table 4 shows the amount of compositing work assigned to each processor for different number of processors. Initially the amount of compositing work per processor increases quite fast; but after 16 processors, it saturates. The last column in Table 4 shows the percentage increase in the amount of compositing work per processor from the previous configuration. As expected, after 64 nodes, the increase is almost negligible. With proper load-balancing, the transformation/rasterization time can decrease linearly with the number of processors, but the compositing time remains the same beyond a certain number of processors. Eventually, the total rendering time is dominated by the compositing time. Therefore, there is an absolute performance limit to the sort-last method, *no matter how many processors are used*. Moreover, this compositing time bound, and consequently, the total rendering time is independent of the 3D data sets. This is the fundamental disadvantage of the sort-last parallel rendering method.

The third component of the total rendering time is the collection time, which is generally small compared to the other two components. There are two possible choices: *linear collection*, in which one node acts as the collector that receives sub-images from all other nodes, and *logarithmic collection*, in which sub-image collection proceeds in a binary tree fashion. Surprisingly for us, linear collection performs better than logarithmic collection in our experiments, because all other nodes can send their sub-images to the collector node simultaneously, whereas in logarithmic collection, synchronization and message preparation at each step incurs more overhead than the performance gain from having more than one node perform the collection.

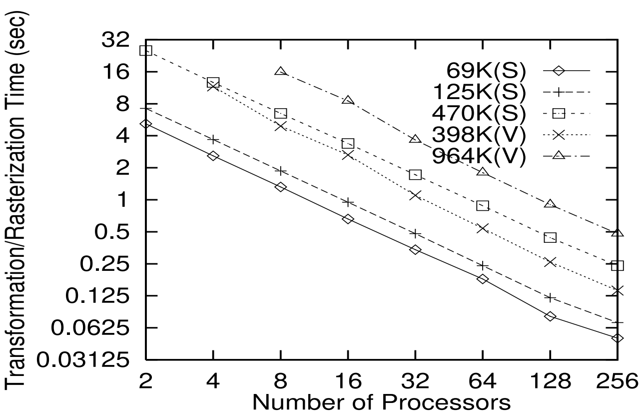

Figure 5. *The transformation/rasterization time in sec for five 3D data sets using different number of processors. The axis for number of nodes is in logarithmic scale*

No. of node	Amount of work	Percentage Increase
2	0.50A	0.00%
4	0.75A	50.00%
8	0.88A	17.33%
16	0.94A	6.82%
32	0.97A	3.09%
64	0.98A	1.03%
128	0.99A	1.02%
256	1.00A	1.01%

Table 4. *The amount of compositing work per processor increases as the number of processors used in transformation/rasterization increases, assuming perfect load balancing. A is the size of the composited images.*

4.3. Comparison Among Compositing Algorithms

Finally, we compare the performance of three compositing algorithms, with respect to the following performance factors: (1) Number of steps: A step is defined as an exchange of sub-images among processors and the associated compositing operation; (2) Number of messages: Each sub-image exchanged is counted as a separate message; (3) Load balancing: At each step, the compositing work assigned to each processor should be as equal to possible.

In the *binary tree compositing* scheme, the number of steps required to merge N images is $log(N)$ and the number of of messages needed is $O(N)$. Both are the theoretical minimum. Unfortunately, as the compositing algorithm progresses, more and more processors become idle, which leads to load imbalance and thus higher total compositing time. As shown in Figure 6, the execution time of binary tree compositing doesn't saturate at large number of processors.

In the *parallel pipeline compositing* scheme, all the processors is given the same amount of work at each step. However, it takes $N - 1$ steps to complete and exchanges $(N - 1) * N$ messages. As shown in Figure 6, even though parallel pipeline compositing uses more steps than binary-tree compositing, it still performs better because the computation parallelism out-weighs the message-passing overhead. Still, the compositing time increases with the number of processors, and it does not saturate even for 256 processors, which should have been the case for an ideal parallel compositing scheme as shown in Figure 6. The curve for the "ideal" scheme is obtained by using the compositing time for 2-processors as the base case and the percentage increase from Table 4.

The *binary swap compositing* scheme exploits the advantages of both binary tree and parallel pipeline. The number of steps is $log(N)$ and at each step and all the processors are equally loaded. The number of messages exchanged is $log(N) * N$, which is much less than the parallel pipeline approach. As shown in Figure 6, the binary swapping compositing time increases initially with more processors, and then remains flat from 64 nodes onwards. This is the ideal behavior as predicted in Table 4 and shown by the "ideal" curve in Figure 6 . Hence binary swap is the best among the three for Intel Paragon machine, and the parameters of our current prototype implementation are set for binary swapping.

To verify that binary swap is indeed close to the ideal parallel compositing scheme, we further decompose the compositing time into comparison time, which corresponds to the arithmetic operations during compositing, and communication time, which corresponds to message transfer due to sub-image exchange. Both components saturate at 64-node configurations, as shown in Table 5. The fact that the trends in Table 5 and 4 are similar to each other, proves that binary swap indeed approximates closely the ideal full parallel compositing scheme.

5. Conclusion

We have developed a fully operational parallel Mesa library using the sort-last strategy that can be linked with parallel applications to provide run-time visualization on large-scale message-passing parallel machines, such as Intel's Paragon. We also developed a unified framework to describe parallel compositing algorithms, and show that the

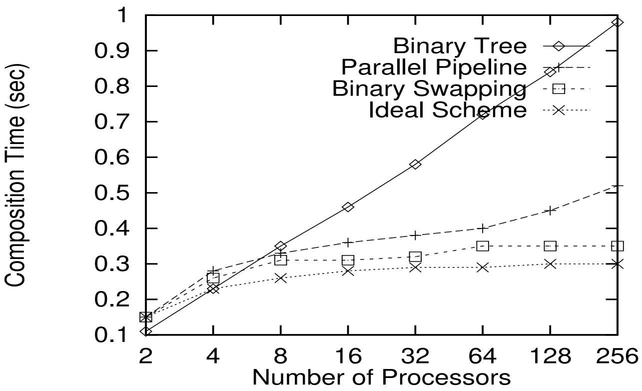

Figure 6. *The compositing time in sec for three compositing schemes using different number of processors.The axis for number of nodes is in logarithmic scale.*

No. of nodes	Comparison time	Communication time
2	0.08	0.07
4	0.11	0.15
8	0.12	0.19
16	0.13	0.18
32	0.14	0.18
64	0.14	0.21
128	0.14	0.21
256	0.14	0.21

Table 5. *The comparison and communication components of the compositing time versus different number of processors.*

two most popular parallel compositing algorithms are just two extreme instances of this family. Given such a framework, one can potentially develop automatic means to tune for the optimal design point in this spectrum that makes the best tradeoff among computation parallelism and communication cost for a given rendering task. Through a detailed performance analysis, we showed that the overall scalability of the parallel Mesa library is quite close to the best one can achieve for the given hardware architecture.

6. Acknowledgement

This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

References

[1] K. Akeley. RealityEngine Graphics. In *SIGGRAPH 1993*, August 1993.

[2] M. Cox. *Algorithms for Parallel Rendering*. PhD thesis, Department of Computer Science, Princeton University, May 1995.

[3] M. Cox and P. Hanrahan. A Distributed Snooping Algorithm for Pixel Merging. *IEEE Parallel and Distributed Technology*, 1994.

[4] T. W. Crockett. Design Considerations for Parallel Graphics Libraries. *ICASE Report No. 94-49*, June 1994.

[5] T. W. Crockett. An Introduction to Parallel Rendering. *Parallel Computing*, 23(7), 1997.

[6] T. W. Crockett and T. Orloff. Parallel Polygon Rendering for Message-passing Architecture. *IEEE Parallel and Distributed Technology: Systems and Applications*, 2(2), 1994.

[7] H. Fuchs et al. Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-enhanced Memories. In *SIGGRAPH 1989*, July 1989.

[8] R. J. Karia. Load Balancing of Parallel Volume Rendering with Scattered Decompositing. In *IEEE Scalable High Performance Computing Conference*, May 1994.

[9] T. Y. Lee et al. Image Composition Schemes for Sort-last Polygon Rendering on 2d Mesh Multicomputers. *IEEE Transactions on Visualization and Computer Graphics*, 2(3), 1996.

[10] K. L. Ma et al. Parallel Volume Rendering using Binary-swap Compositing. *IEEE Computer Graphics and Applications*, 14(4), 1994.

[11] S. Molnar et al. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4), 1994.

[12] S. Molnar, J. Eyles, and J. Poulton. Pixelflow: High-speed Rendering using Image Composition. In *SIGGRAPH 1992*, July 1992.

[13] J. S. Montrym et al. InfiniteReality: a Real-time Graphics System. In *SIGGRAPH 1997*, August 1997.

[14] C. Mueller. Sort-first Rendering Architecture for High-performance Graphics. In *Symposium on Interactive 3D Graphics*, April 1995.

[15] C. Mueller. Hierarchical Graphics Databases in Sort-first. In *IEEE Symposium on Parallel Rendering*, October 1997.

[16] C. Shaw et al. A VLSI Architecture for Image Composition. *Advance in Computer Graphics Hardware, III*, pages 183–199, 1988.