Optimal Processor Allocation for Sort-Last Compositing under BSP-tree Ordering

C.R. Ramakrishnan

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
cram@cs.sunysb.edu

Cláudio T. Silva

Visual and Geometric Computing
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
csilva@watson.ibm.com

ABSTRACT

In this paper, we consider a parallel rendering model that exploits the fundamental distinction between rendering and compositing operations, by assigning processors from specialized pools for each of these operations. Our motivation is to support the parallelization of general scan-line rendering algorithms with minimal effort, basically by supporting a compositing back-end (*i.e.*, a sort-last architecture) that is able to perform user-controlled image composition. Our computational model is based on organizing rendering as well as compositing processors on a BSP-tree, whose internal nodes we call the *compositing tree*. Many known rendering algorithms, such as volumetric ray casting and polygon rendering can be easily parallelized based on the structure of the BSP-tree. In such a framework, it is paramount to minimize the processing power devoted to compositing, by minimizing the number of processors allocated for composition as well as optimizing the individual compositing operations.

In this paper, we address the problems related to the static allocation of processor resources to the compositing tree. In particular, we present an optimal algorithm to allocate compositing operations to compositing processors. We also present techniques to evaluate the compositing operations within each processor using minimum memory while promoting concurrency between computation and communication. We describe the implementation details and provide experimental evidence of the validity of our techniques in practice.

1. INTRODUCTION

A simple way of parallelizing rendering algorithms is to do it at the object-space level: *i.e.*, divide the task of rendering different objects among different rendering processors, and then compose the full images together. A large class of rendering algorithms (although not all), in particular scan-line algorithms, can be parallelized using this strategy. Such parallel rendering architectures, where renderers operate independently until the visibility stage, are called *sort-last* (SL) architectures.[1] A fundamental advantage of SL architecture is the overall simplicity, since it is possible to parallelize a large class of existing rendering algorithms without major modifications. Also, such architectures are less prone to load imbalance, and can be made linearly scalable by using more renderers.[2,3] One shortcoming of SL architectures is that very high bandwidth might be necessary, since a large number of pixels have to be communicated between the rendering and compositing processors. Despite the potential high bandwidth requirements, sort-last has been one of the most used, and successful, parallelization strategies for both volume rendering and polygon rendering, as shown by the several works published in the area .[4–7]

In this paper we propose a general purpose, optimal compositing machinery that can be used as a black box for efficiently parallelizing a large class of sort-last rendering algorithms. We consider sort-last rendering pipelines that are based on separating the rendering processors from the compositing processors, similar to what was proposed previously by Molnar.[2] The techniques described in this paper optimize overall performance ans scalability without sacrificing generality or the ease of adaptability to different renderers. Following Molnar, we propose to use a scan-line approach to image composition, and to execute the operations in a pipeline as to achieve the highest possible frame rate. In fact, our framework inherits most of the salient advantages of Molnar's technique. The two fundamental differences between our pipeline and Molnar's are: (1) instead a fixed network of Z-buffer compositors, our approach uses a user-programmable BSP-tree based composition tree; (2) we use general purpose processors and networks, instead of Molnar's special purpose Z-comparators arranged in a tree.

The work of C.R. Ramakrishnan was partially supported by NSF grants CDA-9504275, CCR-9404921 and CDA-9303181. The work of Cláudio T. Silva was partially supported by Sandia National Labs and the Dept. of Energy Mathematics, Information, and Computer Science Office, by the National Science Foundation (NSF), grant CDA-9626370, and was partially conducted while the author was under a Ph.D. fellowship from CNPq-Brazil.

In our approach, hidden-surface elimination is not performed by Z-buffer alone, but instead by executing a BSP-tree model. This way, we are able to offer extra flexibility, and instead of only providing parallelization of simple depth-buffer scan-line algorithms, we are able to provide a general framework that adds support for true transparency, and general depth-sort scan-line algorithms. In trying to extend the results of Molnar to general purposes parallel machines, we must deal with a processor allocation problem. The basic problem is how to minimize the amount of processing power devoted to the compositing back-end and still provide performance guarantees (*i.e.*, frame rate guarantees) for the user. We propose a solution to this problem in the paper.

In our framework the user defines a BSP-tree in which the leaves correspond to renderers*. Also, the user defines a data structure for each pixel, and a compositing function, that will be applied to each pixel by the internal nodes of the BSP-tree. Given a pool of processors to be used for the execution of the compositing tree, and a minimum required frame rate, our processor allocation algorithm partitions the compositing operations among processors. The partition is chosen so as to satisfy the frame-rate needs using the minimum number of processors. During rendering, the user just needs to provide a viewpoint† (actually, for optimum performance, a sequence of viewpoints, since our algorithm exploits pipelining). Upon *execution* of the compositing tree, messages are sent to the renderers specifying where to send their images, so no prior knowledge of the actual compositing order is necessary on the (user) rendering nodes side. For each viewpoint provided, a *complete* image will be generated, and stored at the processor where the root of the compositing tree is located. The system is fully pipelined, and if no stalls are generated by the renderers, our system guarantees a frame rate at which the user can collect the full images from the root processor.

Summary of Contributions

In this paper we present a mechanism for parallelizing known rendering algorithms automatically and efficiently by optimizing processor resources allocated for image composition. We assign compositing nodes (in a BSP-tree) to processors such that each processor gets a connected set of compositing nodes. Individual processors perform compositions by sequentially evaluating their portion of the compositing tree. By pipelining the computations among the processors, we can hide the latency and achieve high sustained frame rates. In this paper, we propose optimal solutions to perform each of the above operations. In particular:

- We present an optimal linear-time algorithm that assigns compositing operations to processors given a minimum acceptable frame rate (see Section 2).

  The same algorithm can be used to find processor assignments that optimize other parameters, such as latency, overall work and the number of processors.

- We describe techniques to (sequentially) evaluate each segment of the compositing tree so as to minimize the amount of buffering necessary. We also present techniques that trade synchronization for buffering in a straightforward manner (see Section 3).

- We provide experimental evidence to validate our model. The techniques described in this paper were implemented on an MPP machine. Our results indicate that the communication and synchronization costs are negligible. We obtain the maximum possible frame rate for any given assignment of compositing operations, Moreover, the frame rates scale very well with increasing resource allocation (see Sections 4 and 5).

2. OPTIMAL PARTITIONING OF THE COMPOSITING TREE

We can view the BSP tree as an expression tree, with compositing being the only operation. In our model, evaluation of the compositing expression is mapped on to a *tree* of *compositing processes* such that each process evaluates exactly one sub-expression. See Figure 1 for an illustration of such a mapping. The actual ordering of compositing under a BSP-tree depends not only on the position of the nodes, but also on the viewing direction. So, during the execution phase, a specific ordering has to be obeyed. However, given any partition of the tree, each subtree can still be executed independently. Furthermore, evaluation of multiple compositing operations can be effectively pipelined by having the nodes "fire up" in a on-demand fashion. See Section 3 for more details.

*The renderers perform user-defined rendering functions.

†Some algorithms, such as Z-buffer polygon rendering, do not require such a broadcast to the compositing nodes.

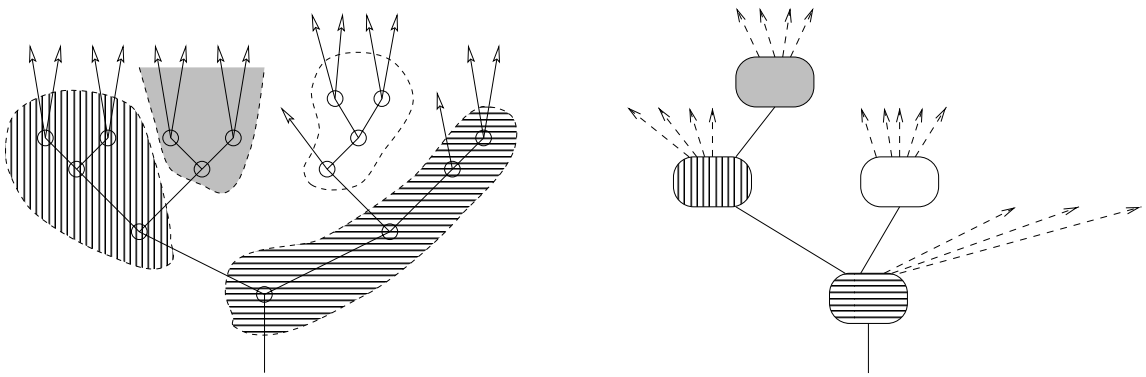

Figure 1. (a) A BSP tree, showing a grouping of compositing operations and (b) the corresponding tree of compositing processes. Each compositing process can be mapped to a different physical node in the parallel machine.

A Cost Model for Compositing

The *weight* of a compositing process is the total time taken to output one composite image. Let the time taken to perform one compositing operation (*i.e.*, compose two full images, or whatever the unit of composition might be) be t_p and the time taken to send or receive one image over the network be t_c. Consider a process that performs k compositing operations. Note that such a process takes $k+1$ input images and composes them into one output image. Hence, the total time taken by the process (ignoring synchronization) to output one image is $kt_p + (k+2)t_c$. Thus, the weight of a compositing process is a linear function of the number of compositing operations performed by the process. The latency of compositing is the maximum weight of any root-to-leaf path in the compositing tree, where the weight of a path is measured by the sum of weights of the nodes in the path. However, note that we can pipeline the computations on the compositing process tree. Hence, in our model, the maximum throughput is dictated by the maximum weight of compositing processes in the tree.

Given a required frame rate f, the maximum number of compositing operations performed in any process, $k_{\max}$ is given by

$$k_{\max} = \left\lfloor \frac{\frac{1}{f} - 2t_c}{t_p + t_c} \right\rfloor$$

It should be noted that the maximum (theoretical) frame rate is obtained when each compositing operation is performed by a separate process. Thus, $f_{\max} \leq \frac{1}{t_p+3t_c}$. Some parallel architectures, such as Intel Paragon, permit computation and communication to be done in parallel. In such cases, the time taken to compose k images is $\max(kt_p, (k+2)t_c)$, and hence $k_{\max} = \lfloor \max(\frac{1}{ft_p}, \frac{1}{ft_c} - 2) \rfloor$.

Since the rendering processes are typically an order of magnitude slower than unit image composition operations, many compositing operations can be sequentialized into a compositing process, without affecting the net frame rate. In particular, if f is the sustained frame rate of the rendering processes, then up to $k_{\max}$ compositing operations can be performed sequentially by a single compositing process. Given a tree of compositing operations there are many ways to map compositing processes to subsets of these operations. The interesting problem, then, is to find a mapping of the tree of compositing operations to the smallest number of compositing processes such that the above weight constraint is satisfied. We now describe techniques to derive such mappings.

Optimal Partitioning

We model the problem of deriving optimal clusters of compositing processes as a tree partitioning problem described below:

Bounded-weight Tree Partitioning: Given a binary tree $G = (V, E)$ and an integer $K \geq 1$, partition V into a *minimum number* of disjoint sets $P_1, P_2, \ldots, P_n$ such that, for every $P_i, 1 \leq i \leq n$ the subgraph of G induced by P_i is connected, and $|P_i| \leq K$.

```
Algorithm partition(u)
    /* The algorithm marks the beginning of partitions in
       the subtree of G rooted at u. If more vertices,
       can be added to the root partition, the algorithm
       returns the size of the root partition.
       Otherwise, the algorithm returns 0. */
1.  if (arity(u) = 2) then    /* u is a binary vertex */
2.      w1 := partition(left_child(u));
3.      w2 := partition(right_child(u));
4.      w := w1 + w2 + 1;
5.      if (w > K) then
6.          if (w1 ≤ w2) then
7.              Mark right_child(u) as start of new partition
8.              w := w1 + 1;
9.          else
10.             Mark left_child(u) as start of new partition
11.             w := w2 + 1;
12. else if (arity(u) = 1) then    /* u is a unary vertex */
13.     w := partition(child(u)) + 1;
14. else    /* u is a leaf */
15.     w := 1;
16. if (w = K) then
17.     Mark u as a start of new partition
18.     return(0);
19. else
20.     return(w);
```

Figure 2. Algorithm *partition*.

Note that since G is a tree, the connectivity requirement ensures that the subgraph induced by P_i is a tree. In our model, G is the given BSP tree, and $K = k_{\max}$, the maximum allowed weight of any compositing process. Each set P_i represents a distinct compositing process, and the partitioning denotes mapping of unit compositing operations to the corresponding compositing process. Note that the restriction that the graph induced on P_i by G be a tree captures the requirement that each compositing process evaluates exactly one sub-expression. The above problem is a special case of the Bounded Component Spanning Forest problem[8] where the graph is restricted to a tree. Although the general problem on graphs is NP-complete, the problem can be solved in polynomial time for trees.[9] Below, we describe an algorithm for finding the optimal solution in linear time.

We use $\mathcal{C}, \mathcal{C}'$ to denote partitions. The partition *induced by* $\mathcal{C}$ on a subtree $G' = (V', E')$ of G is the collection of nonempty sets $P_i \cap V'$ for each $P_i \in \mathcal{C}$. Note that the sets in a partition $\mathcal{C}$ can themselves be arranged in a tree: the root of such a tree is called the *root partition* and is denoted by *root*$(\mathcal{C}, t)$.

Algorithm *partition* appears in Figure 2. Let r be the root of the given tree G, and t_1 and t_2 be subtrees rooted at the children of r. The algorithm computes an optimal partition $\mathcal{C}$ of G by first computing optimal partitions for t_1 and t_2, say $\mathcal{C}_1$ and $\mathcal{C}_2$. The root partition of $\mathcal{C}$ is computed simply as *root*$(\mathcal{C}_1) \cup$ *root*$(\mathcal{C}_2) \cup \{r\}$, provided this set has no more than K elements. It is easy to see that the algorithm is optimal provided such a combination is always possible. If the sizes of *root*$(\mathcal{C}_1)$ and *root*$(\mathcal{C}_2)$ do not permit such a full combination, then the root partition is computed as one of *root*$(\mathcal{C}_1) \cup \{r\}$ or *root*$(\mathcal{C}_2) \cup \{r\}$, or when even these exceed K, as $\{r\}$. The optimality of the algorithm follows from the crucial observation that it computes a best partition with the least number of elements in the root partition.

Practical Considerations

Algorithm *partition* provides a simple way of partitioning a BSP-tree to optimize the use of processors, given a performance requirement in terms of minimum the frame rate. Several issues, including machine architecture bottlenecks, synchronization costs, interconnection bandwidth, and mapping the actual execution to a specific architecture (*e.g.*, a mesh-connected MIMD

machine) were left out of the previous discussion. We now describe how Algorithm *partition* can be readily adapted to account for some of the above issues in practice.

Compositing Granularity: Note that there is nothing in the model that requires that full images be composited and transfered one at a time. Actually, one should take hardware constraints such as memory size and bandwidth limitations into consideration when determining the unit size of work and communication. So, for instance, instead of messages being a full image, it might be better to send a pre-defined number of scan-lines. However, when composing large images, the rendering algorithm must generate the images in scan-line order to avoid prohibitive buffering costs.

Communication Bandwidth: Clearly, the compositing machinary must provide sufficient bandwidth for distributing the images during composition in order to achieve the desired frame rate. Given p processors, each performing k compositing operations, the overall aggregate bandwidth required is proportional to $p(k+2)$. It should be clear that as $k_{\max}$ increases, the actual bandwidth requirement actually decreases (both for the case of a SL-full, as well as a SL-sparse architecture) since as $k_{\max}$ increases the number of processors required decreases. This decrease in bandwidth is due to the fact that compositing computation are performed locally, inside each composite processor, instead of being sent over the network. If one processor performs exactly $k_{\max}$ compositing operations, it needs $k_{\max}+2$ units of bandwidth, as opposed to $3k_{\max}$ when using one processor per compositing operation— a bandwidth savings of almost a factor of three!

The messages in our model, which consist of a number scan lines, tend to be large, implying that our method operates on the best range of the message size versus communication bandwidth curve. For instance, for messages smaller than 100 bytes the Intel Paragon running SUNMOS achieve less than 1 MB/sec bandwidth, while for large messages (i.e., 1MB or larger), it is able to achieve over 160MB/sec. (This is very close to 175MB/sec, which is the peak hardware network performance of the machine.) As will be seen in Section 3, our tree execution method is able to completely hide the communication latency, while still using large messages for its communication.

Latency and Subtree Topology: As will be seen in Section 3, the whole process is pipelined, with a request-based computation strategy. This greatly reduces synchronization overheads. In fact, given enough compositing processors, the overall time is only dependent on the performance of the rendering processors. Also, note that the actual *shape* of the subtree that a given processor gets is irrelevant, since the execution of the tree is completely pipelined.

Architectural Topology Mapping: We do not provide any mechanism for optimizing the mapping from our tree topology to the actual processors in a given architecture. With recent advancements in network technology, it is much less likely that the use of particular communication patterns improve the performance of parallel algorithms substantially. In new architectures, the point-to-point bandwidth in excess of 100–400 MB/sec are not uncommon, while in the old days of the Intel Delta, the bandwidth was merely in the order of 20 MB/sec. Also, network switches, with complex routing schemes, are less likely to make neighbor communication necessary. (Actually, the current trend is not to try to exploit such patterns since new fault-handling and adaptive routers usually make such tricks useless.)

Limitations of the Analytical Cost Model: Even though we can support both SL-full and SL-sparse architecture, our model does not make any distinction of the work that a given compositing processor is performing based on the depth of its compositing nodes. However, the experimental results in Section 5 indicate that this limitation does not seem have any impact on the use of our partitioning technique in practice. Actually, frame-to-frame differences might diminish the concrete advantage of techniques that exploit this additional information.

3. OPTIMAL EVALUATION

In the previous section, we described techniques to partition the set of compositing operations and allocate one processor to each partition, such that the various costs of the compositing pipeline can be minimized. We now describe efficient techniques for performing the compositing operations within each processor.

Space-Optimal Sequential Evaluation of Compositing Trees

Storage is the most critical resource for evaluating a compositing tree. We need 4MB of memory to store an image of size 512×512, assuming 4-bytes each for RGB and α values per pixel. Naive evaluation of a compositing tree with N nodes may require intermediate storage for up to N images. We now describe techniques, adapted from register allocation techniques used

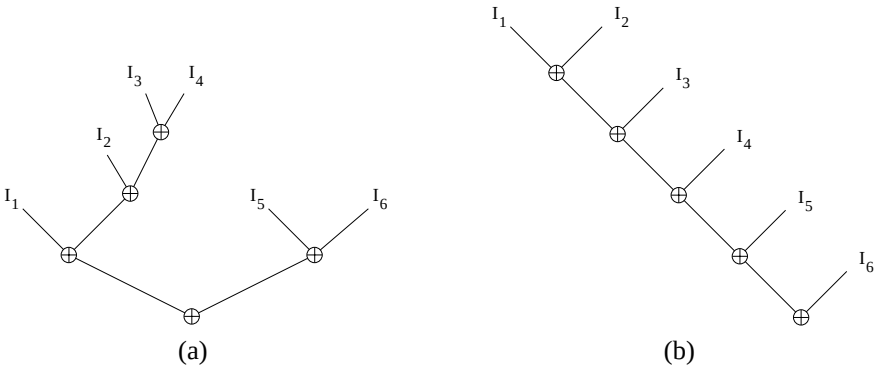

Figure 3. (a) A compositing tree and (b) its corresponding associative tree.

in programming language compilation, to minimize the total intermediate storage required. Figure 3a shows a compositing tree for compositing images I_1 through I_6. We can consider the tree as representing the expression

$$(I_1 \oplus (I_2 \oplus (I_3 \oplus I_4))) \oplus (I_5 \oplus I_6) \tag{1}$$

where $\oplus$ is the compositing operator. Since images I_1 through I_6 are obtained from remote processors, we need to copy these images locally into intermediate buffers before applying the compositing operator. The problem now is to sequence these operations and reuse intermediate buffers such that the total number of buffers needed for evaluating the tree is minimized.

We encounter a very similar problem in a compiler while generating code for expressions. Consider a machine instruction (such as integer addition) that operates only on pairs of registers. Before this operation can be performed on operands stored in the main memory, the operands must be loaded into registers. The number of registers needed to evaluate an expression tree can be minimized, using a simple tree traversal algorithm.[10] Using this algorithm, the compositing tree in Figure 3a can be evaluated using 3 buffers. In general, $O(\log N)$ buffers are needed to evaluate a compositing tree of size N. However, by exploiting the algebraic properties of the operations, we can further reduce the number of buffers needed— to $O(1)$. Since $\oplus$ is associative, evaluating expression (1) is equivalent to evaluating the expression:

$$((((I_1 \oplus I_2) \oplus I_3) \oplus I_4) \oplus I_5) \oplus I_6 \tag{2}$$

The above expression is represented by the compositing tree in Figure 3b, called an *associative tree*.[11] The associative tree can be evaluated using only 2 buffers.

Clearly, any compositing tree can be transformed into a corresponding associative tree. Hence, we can evaluate any compositing tree using only 2 buffers by first transforming it into an associative tree, and evaluating the associative tree. Algorithm *eval_simple* which evaluates a compositing tree with $N \geq 1$ inputs $I_1, I_2, \ldots, I_N$ appears in Figure 4a.

Reducing Synchronization Delays: While Algorithm *eval_simple* evaluates a compositing tree with minimum number of intermediate buffers, it offers no scope to perform communication concurrently with computation. When this algorithm is used to perform the operations associated with each node of the tree of compositing processes, the producer and consumer processes run in lock step, resulting in high synchronization overheads. These overheads can be reduced by using more intermediate storage. For instance, with just one extra buffer, we can issue a read request for the *next* image before performing the current compositing operation, thereby parallelizing communication and computation. A modification of Algorithm *eval_simple* to perform asynchronous communication along these lines yields Algorithm *eval_async* which is given in Figure 4b. In Algorithm *eval_async*, R_1 is used to store the partially composed image. Buffers R_2 and R_3 are used to store the current input image to be composed (R_k, the current buffer) and the next image to be received (R_{5-k}, the pending buffer). Note that, given $k = 2$ or 3, $5 - k = 3$ or 2 respectively, and hence, whenever R_k is the current buffer, R_{5-k} is the pending buffer. The assignment in line 10, $k := 5 - k$, interchanges the current and pending buffers. More aggressive prefetching strategies, which can improve performance at the cost of more buffer space for messages, are also possible.

4. IMPLEMENTATION

In this section, we sketch the implementation of the compositing machinery described in the paper as the compositing back-end of the PVR system.[12] PVR is a high-performance volume rendering system, and it is freely available for research purposes. Our

Algorithm *eval_simple*

1. R_1 := read(I_1);
2. for $j = 2$ to N
3. R_2 := read(I_j);
4. $R_1 := R_1 \oplus R_2$;
5. return(R_1);

(a)

Algorithm *eval_async*

1. read_request(R_1, I_1);
2. read_request(R_2, I_2);
3. wait(R_1);
4. $k := 2$;
5. for $j = 2$ to N
6. if ($j \neq N$) then
7. read_request(R_{5-k}, I_{j+1});
8. wait(R_k);
9. $R_1 := R_1 \oplus R_k$;
10. $k := 5 - k$;
11. return(R_1);

(b)

Figure 4. Algorithms to compose N images (a) with synchronous communication, and (b) with asynchronous communication.

main reason for choosing PVR was that it already supported the notion of separate rendering and compositing clusters.[13] Basically, we replaced the *hard-wired* compositing operation previously implemented in the system by the algorithm in Figure 4b. Initially, before image computation begins, all compositing nodes receive a BSP-tree defining the compositing operations based on the object space partitioning chosen by the user. Each compositing node, in parallel, computes its portion of the compositing tree, and generates a view-independent data structure for its part. Image calculation starts when all nodes receive a sequence of viewpoints.

The rendering nodes simply run the following simple loop:

```
For each (viewpoint v)
  ComputeImage(v);
  p = WaitForToken();
  SendImage(p);
```

Notice that the rendering nodes do not need any explicit knowledge of parallelism; in fact, each node does not even need to know, *a priori*, where its computed image is to be sent. Basically, the object space partitioning and the BSP-tree takes care of all the details of parallelization.

The operation of the compositing nodes is a bit more complicated. First, (for each view) each compositing processor computes (in parallel, using its portion of the compositing tree) an array with indices of the compositing operations assigned to it as a sequence of processor numbers from which it needs to fetch and compose images. The actual execution is basically an implementation of the prefetching scheme proposed in Figure 4b, with each `read_request` being turned into a `PVR_MSG_TOKEN` message, where the value of the token carries its processor id. So, the basic operation of the compositing node is:

```
For each (viewpoint v)
  CompositeImages(v);
  p = WaitForToken();
  SendImage(p);
```

Notice that there is no explicit synchronization point in the algorithm. All the communication happens bottom-up, with requests being sent as early as possible[‡], and speed is determined by the slowest processor in the overall execution, effectively pipelining the computation. Also, one can use as many (or as few) nodes one wants for the compositing tree. That is, the user can determine the rendering performance for a given configuration, and based on the time to composite two images (see compositing capacity in Section 5) it is straightforward simple to scale our compositing back-end for the particular application.

[‡] In PVR, tokens are sent asynchronously, and in most cases, the rendering nodes do not wait for the tokens.

No. of renderers	Compositing processors	Maximum number of compositing operations per processor (K)											
		1	2	3	5	8	13	15	21	44	63	80	100
16	Number	15	10	7	5	2	2	1	–	–	–	–	–
	Time	0.24s	0.24s	0.46s	0.46s	1.56s	1.56s	3.11s	–	–	–	–	–
64	Number	63	42	27	21	9	9	–	4	2	1	–	–
	Time	0.24s	0.26s	0.46s	0.47s	1.35s	1.35s	–	3.80s	6.91s	13.8s	–	–
128	Number	127	85	55	42	18	18	–	9	4	4	2	2
	Time	0.25s	0.26s	0.46s	0.69s	1.57s	1.58s	–	3.13s	7.34s	7.34s	14s	14s

Table 1. Number of compositing processors used and compositing time after first image (to ignore pipeline startup overhead) for a full BSP-tree with 16, 64, or 128 rendering processors.

SL-full versus SL-sparse: Note that one can easily change the SL-full implemented above with a SL-sparse. All the complexity of manipulating sparse images can be localized inside the functions that send and received the images. For maximum performance and flexibility, the rendering algorithms should generate images in scan-line order, and also provide a compact representation for scan-lines (i.e., only the full pixels are represented).

5. PERFORMANCE RESULTS

For our experiments, we used an Intel Paragon XP/S running SUNMOS (installed at Sandia National Laboratories), and we used it in NX compatibility mode. We studied the maximum frame rate that can be achieved with our pipelined evaluation scheme for a given full BSP-tree, and how this frame rate degrades as we increase K (*i.e.*, the maximum number of compositing operations performed on one compositing processor). We ran tests over three different *rendering configurations*: 16, 64 and 128 rendering processors; and several variations on K, leading to several *compositing configurations*. Note that by increasing K from 1 to n (the number of compositing operations) the number of compositing processors decreases from n to 1. Table 1 summarizes the performance on each of these configurations; Figure 5 shows graphically how the compositing time changes as K increases.

In these tests, our primary interest was to study the correlation of K and the compositing time. To achieve the correct effect, we need to make the compositing cluster operate at its maximum speed (for a given K). This was done by making the rendering processors render a single image, and simply send the same image on every subsequent request[§]. This is the scenario where compositing cluster is the bottleneck of the rendering process.

The times reported in the Table 1 are those reported by the PVR collector node, and represent actual wall-clock times. That is, if rendering was fast enough, and the images could be pushed on a frame buffer by the collector, it would be the actual frame rate a user would get. In particular, nothing else needs to be done to the images to prepare them for presentation; in fact, the fully composed image is stored on the node that contain the root of the compositing tree. Note that frame-to-frame coherence does not matter, since we are not exploiting a sparse image representation (these experiments only show the performance of a SL-full architecture).

The *compositing capacity* required of a compositing pipeline is defined as the number of frames that need to be composed per unit time. Note that by varying the number of rendering nodes from 16 to 128, essentially, we make the compositing tree work harder. With 16 rendering nodes producing images at 4 frames per second, we need a compositing capacity of 64 frames per second in our compositing pipeline. The Intel Paragon has very slow processors by today's standards, actually using our image representation (an RGBa image is stored as four floats per pixel), it takes 0.22 seconds to alpha composite two 250 $\times$ 250 images. So, a single processor can composite 4.5 frames per second. To obtain a compositing pipeline with capacity of 64 frames a second needs 14.2 (64/4.5) processors. Hence, in our experimental set up, the compositing pipeline forms the bottleneck even at the lowest rendering speeds (16 rendering nodes at 4 frames a second).

The following observations can be drawn from the data in Table 1:

- As K increases, the frame rates decrease accordingly, since the compositing capacity decreases. Also we can see that our tree partitioning scheme is effective in distributing the load. One can see clearly the finite boundaries where it is possible to save a processor and still achieve the same frame rate (e.g., when K is equal to 8 or 13). Actually with our partitioning algorithm, one can reliably predict the frame rate based (almost) solely on the compositing capacity of a given compositing tree.

[§] In our measurements, this was accomplished by making `ComputeImage` return a pre-computed image immediately upon call.

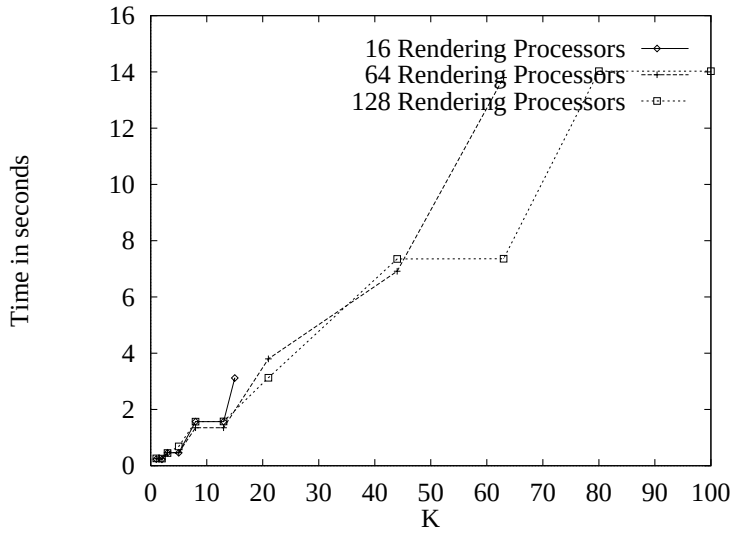

Figure 5. Variation of overall compositing time with K across the three benchmarked configurations. Notice that the compositing capacity needed increases with the number of rendering nodes. In order to keep the desired frame rate, one needs to increase the number of processors allocated to the compositing tree. By keeping K constant, this is achieved automatically, since the number of compositing processors needed also grow, and can be computed by our optimal partitioning algorithm.

- Our asynchronous evaluation of the compositing tree *hides almost all the communication cost*. In fact, the frame rates of the pipeline are independent of its depth. Furthermore, the overall speed of the pipeline is directly related to the maximum number of compositions performed by each node (related to K). For instance, when every processor performs one compositing operation, the frame rate is 4 frames per second (*i.e.*, 0.25s per image), extremely close to the best achievable frame rate of 4.5. As can be seen from the data, it also degrades gracefully.

6. RELATED WORK

Most parallel rendering work (for both geometric and volumetric primitives) on general purpose MIMD machines have used the same processors for both phases. In fact, many techniques have been devised to effectively interleave the two phases on one processor. For example, in volume rendering using the Binary-Swap[7] method, all processes synchronize between rendering and compositing phases as well as during composition. For polygon rendering, the method described by Ellsworth[14] changes states locally between the transformation and the rasterization phases, avoiding global synchronization. In contrast, using processors perform specialized tasks, the rendering and compositing phases can overlap in time, and in fact, can be pipelined.[12,15] Hardware builders have been using dual type configurations for a long time. The distinction between the two categories has been an important factor in the design and implementation of graphics hardware.[16–18]

The major shortcoming with many parallel compositing algorithms designed specifically for polygon rendering[6,14] is the fact that general orderings, such as those needed to support volume rendering or general scan-line algorithms, is not possible. Instead, the BSP-tree[19] approach we follow is very general, and can support a much broader class of algorithms.

The main distinction between the binary swap method and our rendering pipeline is in the allocation rendering and compositing tasks. While the binary swap method divides these tasks in time, our model divides the tasks in processor space. Binary swap based techniques perform dynamic allocation of resources, and hence can adapt to changes in resource requirements over time. In contrast, our technique, with its static allocation, has lower run-time overheads. Moreover, the time taken to completely compose a single image (*i.e.*, latency) using binary swap is largely independent of the number of processors. In our model, however, this time is proportional to the log of the number of processors. However, it should be noted that, our model permits pipelined computation, effectively achieving composition throughputs that are independent of the number of processors used.

In our model, a total of $2p$ messages are exchanged, per image, when p processors are used; binary swap exchanges $p \log p$ messages. Moreover, the communication patterns in our model are simple (tree of processors), resulting in corresponding simplicity of implementation. Moreover, our model supports partitioning of the object space using arbitrary binary trees, with

no additional complexity. Finally, with a small investment in memory (one extra image buffer per processor), we can effectively parallelize communication and computation.

7. CONCLUSION

We proposed an optimal processor allocation technique for parallelizing compositing back-end for sort-last (volumetric as well spatial) rendering systems based on a BSP-tree execution model. The allocation technique can be readily adapted to optimize other system parameters, such as frame rate and latency. We also described techniques to optimize sequential operations of individual compositing processes, as well as to reduce communication overheads and effectively pipeline compositing operations. The techniques are general in the sense that they form the basis for parallelizing existing sort-last rendering algorithms. We implemented the above techniques on a MPP machine, and our experimental results show that (a) the allocation technique significantly lowers resource usage without degrading overall performance and (b) the buffering techniques lead to effective pipelining, leading to high overall frame rates.

We believe sort-last architectures will become even more important with the advent of fast networks and high-performance workstations with 3D graphics support. Such cards are optimized to generate full images, and by using a sort-last architecture, it should be able to build scalable parallel renderers.

A lot of interesting work still remains to be done. For instance, we plan to extend our cost model to argue about the performance of SL-sparse techniques Also, it would be interesting to design mapping algorithms for lower-bandwidth networks which would minimize the overhead of a SL-architecture: for instance, if network switches are being used, possibly one should spread the compositing nodes over several switches. We also plan to investigate static load balancing techniques for the rendering processors, making them match the speed of the compositing cluster.

Acknowledgements

We thank Estie Arkin, Pat Crossno, Owen Kaser, Arie Kaufman, Joseph Mitchell, Dino Pavlakos, Ron Peierls, Steve Skiena and Brian Wylie for enlightening discussions and help during the preparation of this paper. The second author would also like to thank Arie Kaufman for his constant support and encouragement. We thank Sandia National Laboratories, and the Center for Visual Computing at Stony Brook for the use of computing resources.

REFERENCES

1. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification for parallel rendering," *IEEE Computer Graphics and Applications* **14**(4), pp. 23–32, 1994.
2. S. Molnar, "Combining Z-buffer engines for higher-speed rendering," in *Advances in Computer Graphics Hardware III*, pp. 171–182, 1988.
3. S. Molnar, *Image Composition Architectures for Real-Time Image Generation.* Ph.D. thesis, University of North Carolina, Chappel Hill, 1991.
4. T. W. Crockett, "Parallel rendering." In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 34, Supp. 19, A., pages 335–371. Marcel Dekker, 1996. Also available as ICASE Report No. 95-31 (NASA CR-195080), 1996.
5. S. R. Whitman, "A survey of parallel algorithms for graphics and visualization." In International Workshop on High-Performance Computing for Computer Graphics and Visualization, Swansea, United Kingdom, 1995.
6. T. Lee, C. Raghavendra, and J. Nicholas, "Image composition methods for sort-last polygon rendering on 2d mesh architectures," in *IEEE/ACM Parallel Rendering Symposium '95*, pp. 55–62, 1995.
7. K. Ma, J. Painter, C. Hansen, and M. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications* **14**(4), pp. 59–68, 1994.
8. M. Garey and D. Johnson, *Computers and Intractability*, Freeman, San Francisco, 1979.
9. F. O. Hadlock, "Minimum spanning forest of bounded trees," in *Southeastern Conf. on Combinatorics, Graph Theory and Computing*, 1974.
10. A. Aho, R. Sethi, and J. Ullman, *Compilers — Principles, Techniques, and Tools*, Addison Wesley, 1988.
11. R. Sethi and J. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM* **17**(4), 1970.
12. C. Silva, A. Kaufman, and C. Pavlakos, "PVR: High-Performance Volume Rendering," in *IEEE Computational Science and Engineering*, 1996.

13. C. Silva, *Parallel Volume Rendering of Irregular Grids*. Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook, 1996.
14. D. Ellsworth, "A multicomputer polygon rendering algorithm for interactive scientific visualization," in *ACM SIGGRAPH Symposium on Parallel Rendering*, T. Crockett, C. Hansen, and S. Whitman, eds., pp. 43–48, ACM, Nov. 1993.
15. J. Rowlan, E. Lent, N. Gokhale, and S. Bradshaw, "A distributed, parallel, interactive volume rendering package," in *IEEE Visualization '94*, pp. 21–30, 1994.
16. S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-speed rendering using image composition," in *Computer Graphics (SIGGRAPH '92 Proceedings)*, E. E. Catmull, ed., vol. 26, pp. 231–240, July 1992.
17. H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories," in *Computer Graphics (SIGGRAPH '89 Proceedings)*, J. Lane, ed., vol. 23, pp. 79–88, July 1989.
18. K. Akeley, "RealityEngine graphics," in *Computer Graphics (SIGGRAPH '93 Proceedings)*, J. T. Kajiya, ed., vol. 27, pp. 109–116, Aug. 1993.
19. H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures," *Computer Graphics (SIGGRAPH '80 Proceedings)* **14**, pp. 124–133, July 1980.