A Comparison of Parallel Compositing Techniques on Shared Memory Architectures

Erik Reinhard and Charles Hansen

Department of Computer Science
University of Utah
reinhard|hansen@cs.utah.edu

Abstract. Parallel compositing techniques have traditionally focused on distributed memory architectures with communication of pixel values usually being the main bottleneck. On shared memory architectures, communication is handled through memory accesses, obviating the need for explicit communication steps. Shared memory architectures with multiple graphics accelerators provide the capability for parallel rendering while combining the partial results from multiple graphics adaptors requires compositing. For this reason, in this paper shared memory architectures are considered for compositing operations. A number of previously introduced parallel compositing algorithms are compared on a shared memory machine, including the binary swap and parallel pipeline techniques.

1 Introduction

Compositing is a key operation in many applications, including polygon rendering and volume rendering [10]. It is an object-parallel rendering approach, where each renderer produces a full sized image of its sub-set of objects. Combining these images is a simple process, which involves per pixel comparisons of depth values. As these comparisons need to be carried out for all pixels, compositing is an expensive process and is therefore generally unsuitable for interactive applications. However, interactive compositing is possible if the compositing is done in parallel. Both hardware and software solutions have been proposed. An example of a dedicated hardware compositing network is implemented as part of the PixelFlow system, which is configured as a pipeline of compositors [3].

A number of software solutions to parallel compositing have been proposed as well, all working on general purpose distributed memory architectures. These methods include parallel pipeline algorithms [6] and binary swap [8, 4], which was used in the ParVox distributed volume rendering system [7]. The binary swap method was also adapted to work on parallel graphics hardware [11]. Compression of the intermediate images to reduce network bandwidth have also been investigated [1].

A novel compositing scheme for shared memory architectures based upon a similar idea to snooping cache lines for cache coherency was proposed by Cox and Hanrahan [2]. In this method, each processor sends its active pixels to the frame-buffer while the other processors *snoop* the pixels written. When one of the pixels written to the frame buffer is closer to the viewer, in Z, than the corresponding active stored pixel, that active stored pixel is marked as inactive or invalidated. Thus, as each processor stores its set of active pixels, other processors reduce their active pixel sets, thereby reducing the bandwidth requirement for the composite of all images.

Distributed memory compositing has a number of significant drawbacks. First, the amount of data involved in the compositing process is large. Therefore, the amount of data communication between processors is large. While compression is one technique for reducing the amount of data [1], the compression can increase the latency during the compositing phase of rendering. The communication delays reduce the scalability and efficiency of parallel compositing strategies. Second, after the parallel compositing stage is completed, each processor holds a completed sub-image. These sub-images need to be collated to produce the final result which can be displayed.

Shared memory architectures, such as the Silicon Graphics Origin 2000, offer parallel graphics hardware in conjunction with general purpose parallelism. These machines are ideal for parallel polygon rendering, utilizing the multiple graphics adaptors in parallel. The compositing of the results can then be done in software, overlapping rendering with compositing. The rendering step will scale linearly with the number of graphics pipelines, while the compositing step should scale well with the number of general purpose processors available. Additionally, having the ability to divide scalar fields among the graphics adaptors can allow interactive parallel volume rendering. This technique also requires compositing of the partial images obtained from the graphics pipelines although the compositing operations are blended rather than conditionally combined as in the polygon case.

For these reasons, such shared memory architectures may provide a useful alternative as a substrate for compositing. As communication is achieved via memory, all images that need to be composited into one, can be read by all processors. This avoids extensive data communication and for this reason, the relative performance of various parallel compositing strategies, as reported for distributed memory machines, may turn out to be different. Secondly, this architecture allows the compositing process to produce the final result without an extra collation step, thereby increasing frame rates. In this paper, the compositing algorithms mentioned above are implemented on a shared memory architecture and their performances are assessed. We also compare these algorithms with a straightforward compositor which exploits the advantages of shared memory directly (Section 4).

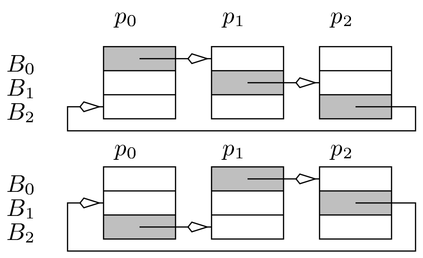
The following sections outline the parallel compositing algorithms and indicate the differences between distributed and shared memory implementations (Sections 2 to 4). A discussion of their time complexities and the implications on interactive rendering are discussed in section 5. The renderer which was used as a basis for these compositing algorithms is detailed in Section 6, while test results are presented in Section 7, followed by conclusions in the final section.

2 Parallel pipeline

Originally proposed for mesh connected distributed memory machines, the parallel pipeline algorithm organizes the processors in a ring. The images and z-buffers which are to be composited are divided into P sub-images, which equals the number of processors. These sub-images flow around the ring through each processor in $P-1$ stages, each consisting of a compositing and a communication step. The distributed memory algorithm is detailed in Figure 1.

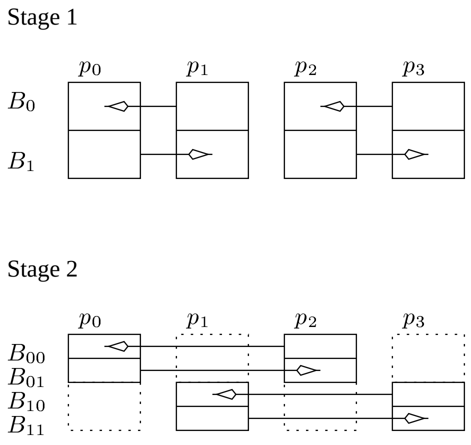
Adapting this algorithm to run on shared memory computers is straightforward. The send and receive instructions are not necessary, as memory can be read directly. Therefore, in each of the $P-1$ stages, each processor composites its sub-image k with the same sub-image that belongs with processor p_k.

By replacing the last iteration of the compositing algorithm with a compositing step which writes the result into a contiguous block of memory, the final collation of sub-images, necessary in distributed memory solutions, can be avoided. The result can be



```
do in parallel
    let $B$ = sub-image from $\frac{iH}{P}$
        to $\frac{(i+1)H}{P}$
    for $j = 1, \ldots, P-1$
        send $B$ to $p_{next}$
        let $k = (i-j)$ mod $P$
        receive $B'_k$ from $p_{prev}$
        compose $B'_k$ with $B$
        let $B = B_k$
```

Fig. 1. Parallel pipeline compositing on distributed memory architectures (after [6]). Left: graphical representation. Right: pseudo-code. The variables are coded as follows: P is the total number of processors, H is total image height, B and Z are sub-images and p_i is processor number i.



```
do in parallel
  let $h = 0$
  let $height = H$
  let $level = \log_2 P$
  for $l = 0, \ldots, levels - 1$
     let $B$ = sub-image from
         $h$ to $h + height$
     if ($p_i$ & $2^l$)
         send lower half of $B$ to $p_{i-2^l}$
         receive upper half from $p_{i-2^l}$
         composite upper half of $B$
         let $h = h + height/2$
     else
         send upper half of $B$ to $p_{i+2^l}$
         receive lower half from $p_{i+2^l}$
         composite lower half of $B$
     let $height = height/2$
```

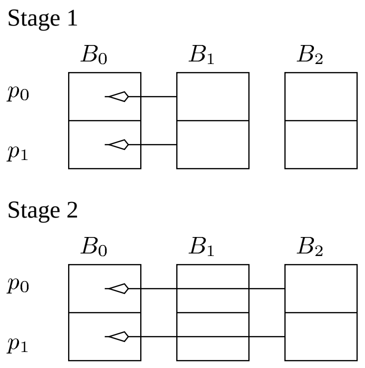
Fig. 2. Binary swap compositing on distributed memory architectures (after [11]). Left: graphical representation. Right: pseudo-code.

displayed directly.

3 Binary swap

The binary swap algorithm was originally implemented for CM-5 and T3D distributed memory machines [8, 4]. Later, it was adapted to utilize graphics hardware [11]. The distributed memory algorithm is depicted in Figure 2. Given P images to be composited using P processors, the algorithm completes the compositing task in $\log P$ iterations. In the first iteration, the images to be composited are subdivided into an upper and a lower half. Each processor swaps half its image for half the image held by a neighboring processor. Both halves are then composited in tandem.

In the next iteration, the half image that a processor just finished compositing is recursively subdivided into two sub-images. One sub-image is swapped with the next-nearest neighbor and both processors continue compositing their sub-images. After $\log P$ iterations, each processor holds a fully composited sub-image. After collating all partial images, the result can be displayed.



```
do in parallel
    let $B =$ sub-image $i$ for processor $p_i$
    for $n = 1, \ldots, P$
        composite sub-images $B_0$ and $B_n$
```

Fig. 3. Shared memory compositing. Left: graphical representation showing how three sub-images are composited using two processors. Right: pseudo-code.

The transition to a shared memory implementation is accomplished by removing the communication steps from the algorithm, as each processor can read another processor's sub-image directly. Otherwise, the shared memory binary swap algorithm is identical to its distributed memory variant. As explained in the previous section, this algorithm also allows the last iteration to be replaced with a step which writes the result into a single contiguous block of memory, avoiding the separate collation step.

4 A shared memory compositor

As separate communication steps between processors are absent in shared memory applications, a straightforward parallel compositing scheme would be to subdivide each partial image into P sub-images and have each of the P processors composite one of these sub-images [2]. Pseudo-code and an example of three sub-images being composited by two processors is given in Figure 3.

On distributed memory machines this mechanism has been used as well. It proceeds by first sending all partial results for a given pixel to the processor that manages that pixel. Then as partial results are received, the result is composited. On distributed memory architectures this method is known as 'Direct Send' and was used for example in [5] and [9].

Note that the result is available in a single block of memory, allowing direct display of the result. As with the algorithms described in the previous two sections, collation of the results is therefore not necessary and the image can be displayed directly.

5 Time complexity

In this section the theoretical time complexity of the above three compositing algorithms is discussed, along with its implications for interactive rendering. For the moment we assume that the number of images to be composited equals the number of processors. Also, it is assumed that the number of images/processors is 2 or larger.

The shared memory binary swap algorithm iterates $\log_2 P$ times and during each iteration each processor composites $N/(2^i)$ pixels (where N is the number of pixels in the image and i is the iteration number). The total number of pixels each processor composites can therefore be expressed as a geometric series with the following solution:

$$\sum_{i=1}^{\log_2 P} \frac{N}{2^i} = N \sum_{i=0}^{\log_2 P} \left(\frac{1}{2}\right)^i - N = N\left(2 - \frac{2}{2^{1+\log_2 P}}\right) - N = N(1 - \frac{1}{P}) \quad (1)$$

The binary swap compositing method therefore has a time complexity of $O(N)$. The parallel pipeline algorithm requires $P-1$ iterations. Each processor composites N/P pixels during each iteration, so that the time complexity of this algorithm is $O(N)$ (where N is the number of pixels in the image):

$$(P-1)\frac{N}{P} = N(1 - \frac{1}{P}) \quad (2)$$

The time complexity for the compositor described in section 4 is $O(N)$ as well, because each processor also iterates though $P-1$ steps, compositing N/P pixels at each step.

Because all three algorithms have the same time complexity, the differences in performance are due to the differences in overhead and cache efficiency. None of the algorithms cause any two processors to read or write the same pixels at the same time, so memory clashes will not occur. As a result, we expect the simplest algorithm to perform best, i.e. the compositor algorithm presented in the previous section is expected to be slightly more efficient than either the binary swap or parallel pipeline algorithms.

Note that for all three algorithms, the time complexity is independent of the number compositors. This is due to the assumption that the number of images to be composited into one is equal to the number of processors that perform the actual compositing. Under this assumption, if the number of renderers is increased, the number of compositors should also be increased by the same number. The rendering step will then become cheaper, while the compositing step remains equally expensive. Hence, as long as the number of compositors equals the number of renderers, there will be an optimum number of renderers/compositors. Deviating from this optimum will either cause the renderers or the compositors to become the bottleneck.

In order to reduce the compositing time, the number of compositors needs to be increased *relative* to the number of renderers. For a given number of renderers, doubling the number of compositing processors should roughly half the compositing time, resulting in (near) linear speed-up for the compositing step, enabling interactive execution. After describing the renderer in more detail in the following section, these claims are assessed in Section 7.

6 Renderer and compositor implementation

The platform used for this research is a Silicon Graphics Origin 2000 with 32 processors and 8 graphics pipes. The rendering step uses between two and eight of the available graphics engines to render partial images to be composited. The threads implementing hardware polygon rendering were placed with processors located near the graphics hardware, which minimizes data transfer through the machine. However, it should be noted that within our implementation, the performance benefit of this placement is under 1%.

Each renderer renders only a subset of the input polygons, hence increasing the number of graphics pipes should linearly improve rendering performance assuming that the polygons are evenly distributed within the viewing frustum and with similar depth complexity among the different graphics pipes. OpenGL was used to access the graphics hardware.

Compositing is achieved in software and utilizes processors that are not involved in rendering. The placement of compositing processes on these processors is otherwise arbitrary. Because the Origin 2000 uses a hypercube interconnect, the binary swap algorithm, which was conceived to work on such architectures, was thought to benefit from careful placement of processes on processors. The resulting performance increase we measured for eight compositors is around 1%. However, for larger experiments it is not possible to map both the renderers and the compositors optimally. As the benefits are small for either optimisation, we have chosen the map the renderers near the graphics hardware.

Double buffering is used for the partial images generated by the renderer to ensure that the compositors operate upon finished partial images. The result of the compositing step is also double buffered to provide a smooth display. Hence, the time lag between user input and displaying the result is three frames. Between successive frames, all renderers and compositors synchronize.

7 Results

Test results were obtained using two different scenes, which are depicted in Figure 4. The plants model consists of $200,320$ triangles and the head is modeled by $396,787$ triangles. An example of the partial images generated by four different graphics pipes is given for both scenes in Figure 5. All images are rendered at a resolution of either 512^2 or 1024^2 pixels, and each of the following bench-marks consists of 100 frames. Timings provided are in seconds and 'total time' refers to the time taken for the system to render and composite 100 frames. 'Compositing time' refers to the time taken to just composite 100 frames.

Bench-marks were carried out for different numbers of renderers and compositors. First, rendering time and compositing time are compared for the shared memory, binary swap and parallel pipeline algorithms. The results are presented in Figures 6 and 7. These figures show that for the head test scene, smaller images consisting of 512^2 pixels can be composited as fast as they can be rendered if the number of compositors is greater than or equal to the number of renderers. Hence, for this test scene combined with smaller images, the renderers were always the bottleneck.

The compositing part for these tests appear to scale super-linearly with the number of compositors. This is explained by the fact that subdivided 512^2 images are small enough to largely fit into the caching structure of the Origin 2000. Better utilization of the memory hierarchy (a reduction in cache misses) is achieved when using more compositors (each compositing a smaller sub-image). This behavior is not observed for larger images (as shown in Figure 7).

For test images generated at a resolution of 1024^2 pixels, the number of compositors needs to be roughly four times higher than the number of renderers (this is consistent with the fact that four times more pixels need to be composited than with 512^2 images). If fewer compositors are used, compositing becomes the bottleneck. If more compositors are used, the renderers become the limiting factor. However, this factor of four is scene dependent, as the rendering performance of the graphics pipes is nearly linear in the number of triangles rendered. The timings do not differ significantly between the three different compositing schemes. This is in line with their theoretical time complexities as outlined in section 5.

Fig. 4. Plants (200,320 triangles) and head model (396,787 triangles).

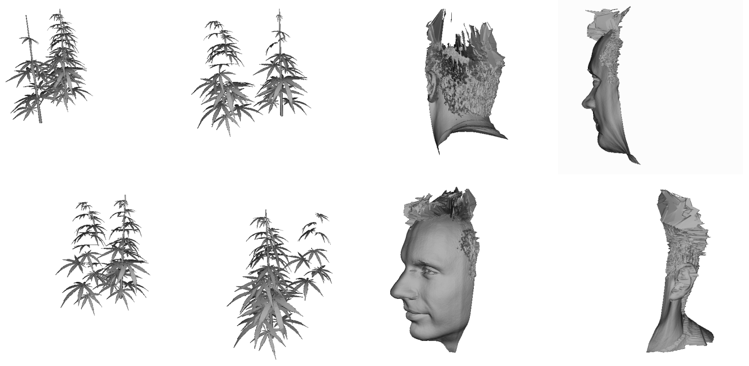


Fig. 5. Partial images generated by four graphics pipes for the plants (left) and head scenes (right).

Although the plants test scene is roughly half the size of the heads test scene, our measurements show similar behavior for both scenes. By plotting the total time and compositing time for 512^2 images for both scenes (Figure 8), the near linear scalability of the rendering step is confirmed. Figure 9 shows that the compositing component is indeed scene independent. Both results are according to expectation.

8 Conclusions

The differences in timing between the three compositing algorithms are very small, as predicted in section 5. Although different parallel compositing techniques have their merit in distributed memory environments, shared memory architectures appear to be quite insensitive to the different orders in which pixels are read and written. All three compositing techniques never read the same memory location by more than one processor at the same time. This appears to be sufficient to guarantee efficient execution.

All three shared memory algorithms have the additional advantage that collation of

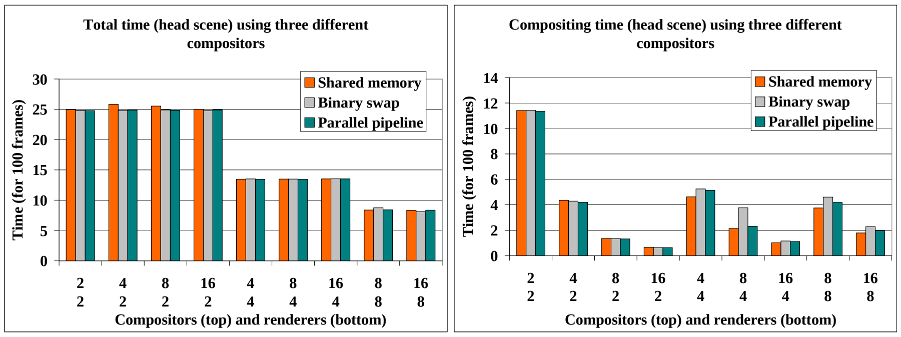


Fig. 6. Total time and compositing time for the head scene, using a resolution of 512^2 pixels.

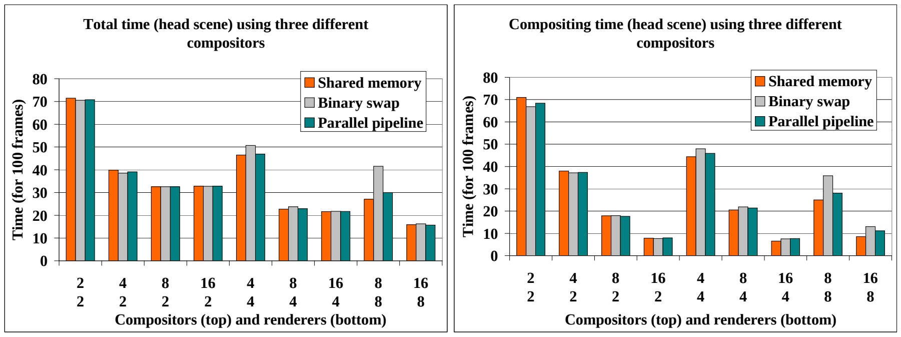


Fig. 7. Total time and compositing time for the head scene, using a resolution of 1024^2 pixels.

results can be merged with the last iteration of the compositing algorithm, obviating the need for a separate and potentially expensive collation step.

Acknowledgments

Thanks to Steve Parker for his efficient threads library and to Tushar Udeshi for his hardware rendering code. This work was supported by NSF grants NSF-9977218 and NSF-9978099 and by the DOE Advanced Visualization Technology Center.

References

1. J. Ahrens and J. Painter, *Efficient sort-last rendering using compression-based image compositing*, in Second Eurographics Workshop on Parallel Graphics and Visualization, Eurographics, 1998, pp. 145–151. Rennes.
2. M. Cox and P. Hanrahan, *Pixel merging for object-parallel rendering: a distributed snooping algorithm*, in ACM SIGGRAPH Symposium on Parallel Rendering, T. Crockett, C. Hansen, and S. Whitman, eds., ACM, Nov. 1993, pp. 49–56.
3. J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover, *PixelFlow: The realization*, in 1997 SIGGRAPH / Eurographics Workshop

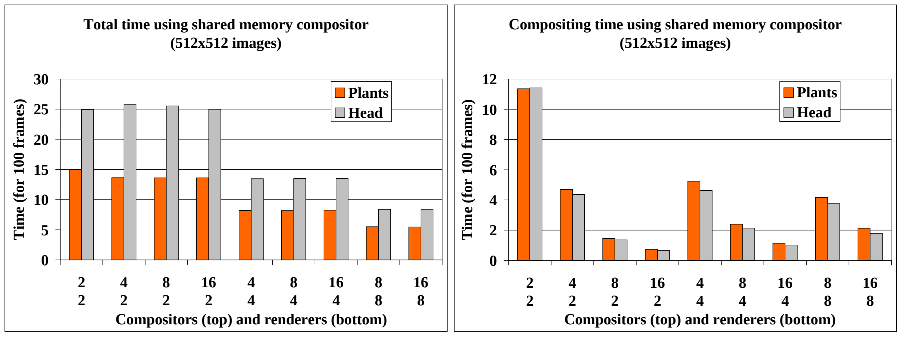


Fig. 8. Total time (left) and compositing time (right) for 512^2 images.

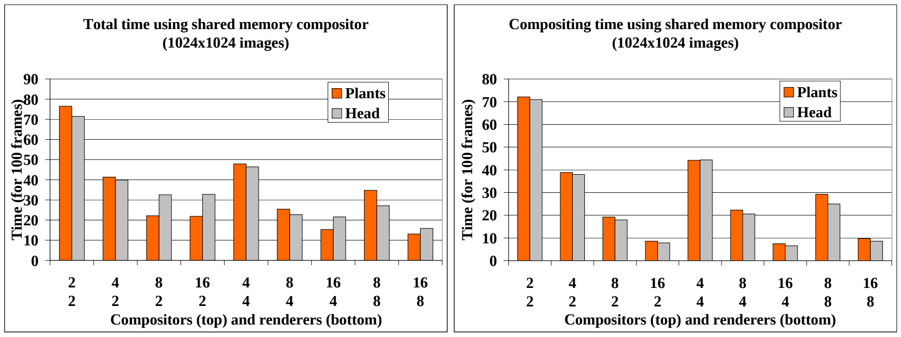


Fig. 9. Total time (left) and compositing time (right) for 1024^2 images.

on Graphics Hardware, S. Molnar and B.-O. Schneider, eds., New York City, NY, Aug. 1997, ACM SIGGRAPH / Eurographics, ACM Press, pp. 57–68.

4. C. Hansen, M. Krogh, J. Painter, G. C. de Verdiere, and R. Troutman, *Binary-swap volumetric rendering on the T3D*, in Cray Users Group Conference, Denver, Mar. 1995.
5. W. M. Hsu, *Segmented ray casting for data parallel volume rendering*, in ACM SIGGRAPH Symposium on Parallel Rendering, ACM, Nov. 1993, pp. 6–14.
6. T.-Y. Lee, C. Raghavendra, and J. B. Nicholas, *Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers*, IEEE Transactions on Visualization and Computer Graphics, 2 (1996).
7. P. P. Li, S. Whitman, R. Mendoza, and J. Tsiao, *ParVox - A parallel splatting volume rendering system for distributed visualization*, in IEEE Symposium on Parallel Rendering, J. Painter, G. Stoll, and Kwan-Liu Ma, eds., IEEE, Nov. 1997, pp. 7–14.
8. K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, *Parallel volume rendering using binary-swap compositing*, IEEE Computer Graphics and Applications, 14 (1994).
9. U. Neumann, *Parallel volume-rendering algorithm performance on mesh-connected multicomputers*, in ACM SIGGRAPH Symposium on Parallel Rendering, T. Crockett, C. Hansen, and S. Whitman, eds., ACM, Nov. 1993, pp. 97–104.
10. T. Porter and T. Duff, *Compositing digital images*, in Computer Graphics (SIGGRAPH '84 Proceedings), H. Christiansen, ed., vol. 18, July 1984, pp. 253–259.
11. T. Udeshi and C. Hansen, *Parallel multipipe rendering for very large isosurface visualization*, in EUROGRAPHICS - IEEE TCCG Symposium on Visualization, May 1999, pp. 99–108.