The Journal of Supercomputing, 18, 201–220, 2001

Efficient Compositing Methods for the Sort-Last-Sparse Parallel Volume Rendering System on Distributed Memory Multicomputers

DON-LIN YANG dlyang@fcu.edu.tw
JEN-CHIH YU jcyu@fcu.edu.tw
YEH-CHING CHUNG ychung@fcu.edu.tw

Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407

Final version accepted September 15, 2000

Abstract. In the sort-last-sparse parallel volume rendering system on distributed memory multicomputers, one can achieve a very good performance improvement in the rendering phase by increasing the number of processors. This is because each processor can render images locally without communicating with other processors. However, in the compositing phase, a processor has to exchange local images with other processors. When the number of processors exceeds a threshold, the image compositing time becomes a bottleneck. In this paper, we propose three compositing methods to efficiently reduce the compositing time in parallel volume rendering. They are the binary-swap with bounding rectangle (BSBR) method, the binary-swap with run-length encoding and static load-balancing (BSLC) method, and the binary-swap with bounding rectangle and run-length encoding (BSBRC) method. The proposed methods were implemented on an SP2 parallel machine along with the binary-swap compositing method. The experimental results show that the BSBRC method has the best performance among these four methods.

Keywords: sort-last-sparse, parallel volume rendering, image compositing, run-length encoding

1. Introduction

Volume visualization is a well-known methodology for exploring the inner structure and complex behavior of three-dimensional volumetric objects. Existing volume visualization algorithms are commonly divided into two categories, surface rendering and (direct) volume rendering. Surface rendering extracts a given volume data to form a contour surface with a constant-field value and renders the contour surface geometrically. Volume rendering projects the entire volume data semi-transparently onto a two-dimensional image without the aid of intermediate geometrical representations. It is important for users to interactively explore the volume data in real time. However, both surface rendering and volume rendering of large volumes of data are still time consuming, and it is difficult to reach the interactive rendering rate on a single processor.

To achieve the goal of interactive volume visualization, parallel rendering is a very good approach. Molnar *et al.* [12] classified parallel rendering into three categories, sort-first, sort-middle, and sort-last. Among them, the sort-last scheme is the most commonly used in parallel rendering. There are three phases, the partitioning

phase, the rendering phase, and the compositing phase in a sort-last parallel volume rendering system, as shown in Figure 1. In the partitioning phase, a processor partitions the entire volume data into several subvolume data, and distributes these subvolume data to other processors. In the rendering phase, each processor renders the assigned subvolume data into a 2D subimage using either volume rendering or surface rendering algorithms. In the compositing phase, the subimages of processors are composed into a full image using compositing algorithms. The image is then displayed on the screen or is saved as an image file.

A number of parallel volume rendering systems in the sort-last class have been proposed in the literature. Most of the algorithms are implemented on MIMD/SIMD distributed memory multiprocessor systems. In the rendering phase, there are several volume visualization algorithms that can be used, such as the March cube algorithm [10] for surface rendering, and the ray tracing [9], shear-warp [7], and splatting [15] algorithms for volume rendering. In Figure 1, we can see that each processor renders its local subvolume data without communicating with other processors. It can get a good speedup as the number of processors increases. However, a speedup is restricted to certain thresholds because of exchanging subimages with other processors in the compositing phase. This indicates that the compositing phase is a bottleneck in a sort-last parallel volume rendering system when the number of processors exceeds a threshold.

In the sort-last class, the implementation of the image compositing can be divided into two categories, full-frame merging and sparse merging [12]. In the full-frame

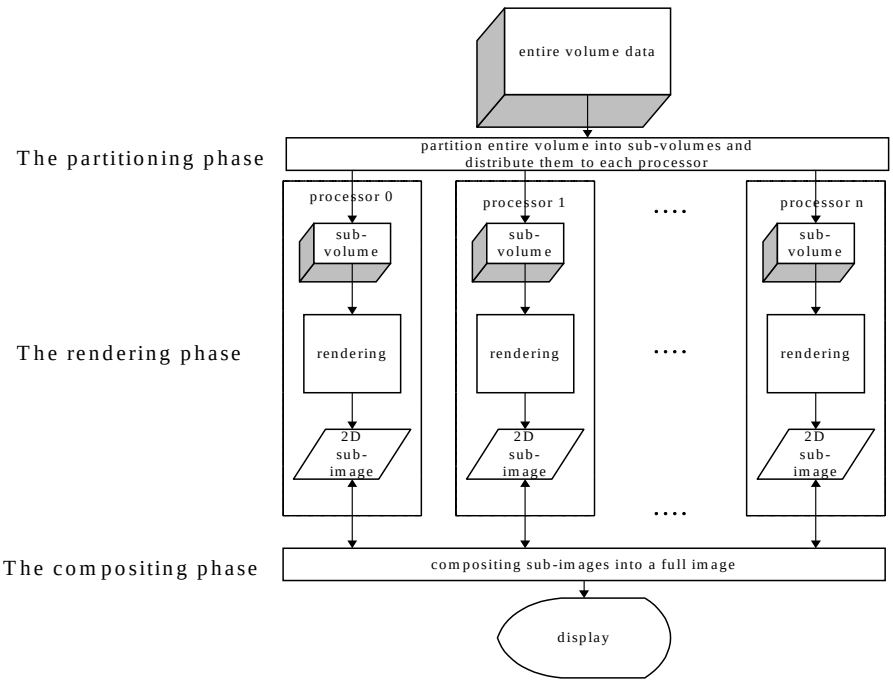

Figure 1. Three phases in a sort-last parallel volume rendering system.

merging implementation, processors exchange full 2D subimage frames without considering the contents of the frames. The full-frame merging is very regular and can be easily implemented in software and hardware, but it is not efficient if the contents of the 2D subimage frames are sparse. On the other hand, in the sparse merging implementation, processors exchange non-blank pixels of the 2D subimage frames. This implementation is more complicated than the full-frame merging implementation. However, it can reduce the communication and the computation overheads in the compositing phase when the 2D subimage frames are sparse.

In this paper, we propose three compositing methods, the binary-swap with bounding rectangle (BSBR) method, the binary-swap with run-length encoding and static load-balancing (BSLC) method, and the binary-swap with bounding rectangle and run-length encoding (BSBRC) method to efficiently reduce the image compositing time in the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. Throughout the paper we will use the terms sort-last-sparse and sort-last sparse merging interchangeably. The proposed methods were implemented on an IBM SP2 parallel machine along with the binary-swap compositing method [11]. Experimental results show that the BSBRC method has the best performance among the four methods.

The rest of the paper is organized as follows. The related work of parallel compositing methods will be given in Section 2. In Section 3, the proposed methods will be described and analyzed in detail. In Section 4, the experimental results of the proposed methods will be presented. Section 5 concludes the paper and discusses some future work.

2. Related work

Many parallel volume rendering systems in the sort-last class have been proposed for distributed memory multiprocessor systems. In the image compositing phase, the compositing methods can be divided into two cases, the buffered case and the sequenced case [14]. In the buffered case, each processor is responsible for handling a fixed portion of the image. Each processor allocates a buffer and receives pixels in the same fixed portion of the image from other processors once. After compositing pixels in the buffer, each processor generates the final image of the portion it handles. In the buffered case, each processor has to send and receive $n - 1$ messages at the same time. Hsu [4] and Neumann [14] used the buffered case method to composite subimages into a final image. In the sequenced case, such as the tiling approach [6], parallel pipeline [8], binary-tree [1], and binary-swap [11], each processor receives a message from one processor and composites received pixels immediately in each compositing stage. Each processor repeats the same step in each compositing stage until it generates the portion of the final image. Since our methods are based on the binary-swap compositing method, we will describe it in detail in the next subsection.

The compositing methods described above can be applied to the sort-last-sparse or the sort-last-full implementations. Some methods for the sort-last-sparse parallel

volume rendering system have been proposed in the literature [1, 2, 8, 11]. In the following section, we will briefly describe them.

Ahrens and Painter [1] proposed a compression-based image compositing algorithm. They used a lossless compression technique, run-length encoding [3], to compress non-blank pixels. They applied this scheme to the binary-tree compositing method for parallel surface rendering. In the surface rendering, a pixel value is represented using red, blue, green, depth, and count fields. The algorithm initially uses the first pixel as the base pixel and then compares it with the next pixel. Iterating through the pixels of the image by column or row, the algorithm compares the base pixel with the current pixel. If the values of red, blue, green, and depth of the two pixels are equal, the value of the count field in the base pixel is increased by one. Otherwise the base pixel is the encoded pixel. The base pixel value is then set to the current pixel value and the value of its count field is set to 1. In the compression phase, the time complexity of the algorithm is $O(n)$, where n is the number of pixels of the input image. In the phase of compositing compressed images, two images are input and one result image is output. There are two cases in this phase. One case is that one input image contains a *run* (an encoded pixel) and the other does not. In this case, an output pixel's value is extracted from the run and the count field in the run is decreased by 1. The other case is that both input images contain runs. The runs of pixels can be composited together. The length of runs to be composited is equal to the smaller count value of the two runs. The value of the count field of the run with a larger count value is set to its count value minus the count value of the other run. The comparison continues until there are no more runs. In the compositing phase, the time complexity of the algorithm is $O(n)$ for the worst case and is $O(1)$ for the best case.

Lee [8] proposed a direct pixel forwarding method and applied it to the parallel pipeline compositing algorithm for the sort-last-sparse polygon rendering. In the direct pixel forwarding method, explicit information is used to locate non-blank pixel positions in a subimage. For each non-blank pixel, its value is represented by red, blue, green, depth, and its x and y coordinates of the pixel. In the compositing phase, each processor only composites non-blank pixels and stores the result pixels in correct positions according to the x and y coordinates of pixels. Cox and Hanrahan [2] also applied this scheme to a distributed snooping algorithm for polygon rendering.

Molnar *et al.* [12] indicated that the sort-last sparse merging methods are load unbalanced if one processor sends more non-blank pixels than other processors. To solve the load imbalance problem, one can assign each processor an interleaved array of non-blank pixels such that each processor sends an almost equal number of pixels to other processors. Lee [8] applied this scheme to the parallel pipeline compositing algorithm with direct pixel forwarding.

To avoid sending blank pixels and to reduce overheads of processing non-blank pixels, the bounding rectangle [3] is a good choice. Iterating through the pixels in an input image, the bounding rectangle scheme records the coordinates of the upper-left and the lower-right corners of the bounding rectangle. In the compositing phase, each processor only handles pixels in the bounding rectangle. Lee [8] applied the bounding rectangle scheme to the parallel pipeline compositing algorithm.

Ma *et al.* [11] also used a bounding rectangle to cover all non-blank pixels at each compositing stage.

3. The proposed compositing methods

In this section, we will first describe the binary-swap (BS) compositing method proposed by Ma *et al.* [11]. Then we will explain the proposed compositing methods, the binary-swap with bounding rectangle (BSBR) method, the binary-swap with run-length encoding and static load-balancing (BSLC) method, and the binary-swap with bounding rectangle and run-length encoding (BSBRC) method, in detail.

In order to derive the cost of the compositing phase in terms of processing time, a summary of the notations used in this paper is introduced here.

- $T_{comp}(L)$—Computation time of method L.
- $T_{comm}(L)$—Communication time of method L.
- T_s—Start-up time per message.
- T_c—Message transmission time per byte.
- T_o—Computation time of "over" operation per pixel.
- A—Image size in pixels, $A^{1/2} \times A^{1/2}$.
- P—Number of processors (PE).

3.1. The binary-swap compositing method

The binary-swap compositing method [11] was originally proposed for parallel volume rendering to composite ray-traced subimages to a full image. The key idea is that, at each compositing stage, two processors are paired. One processor in a pair exchanges half of its subimage with that of the other. After exchanging subimages, each processor composites the half image that it keeps with the received half image by using the *over* operation. Figure 2 illustrates the binary-swap compositing method using four processors. The algorithm of the binary-swap compositing method is shown in Figure 3. In Figure 3, the terms of PE and PE′ are the paired processors.

In the binary-swap compositing method, $\log P$ communication steps are required. When the compositing of subimages to a full image is completed, the total number of pixels transmitted is $P \times \sum_{k=1}^{\log P} A/2^k$, and each pixel consists of intensity and opacity. Each pixel is represented by 16 bytes. Therefore, for each processor, the local computation time and the communication time in the binary-swap compositing method are

$$T_{comp}(BS) = \sum_{k=1}^{\log P} \left(T_o \times \frac{A}{2^k} \right) \text{ and} \tag{1}$$

$$T_{comm}(BS) = \sum_{k=1}^{\log P} \left(T_s + \left(16 \cdot \frac{A}{2^k} \right) \times T_c \right). \tag{2}$$

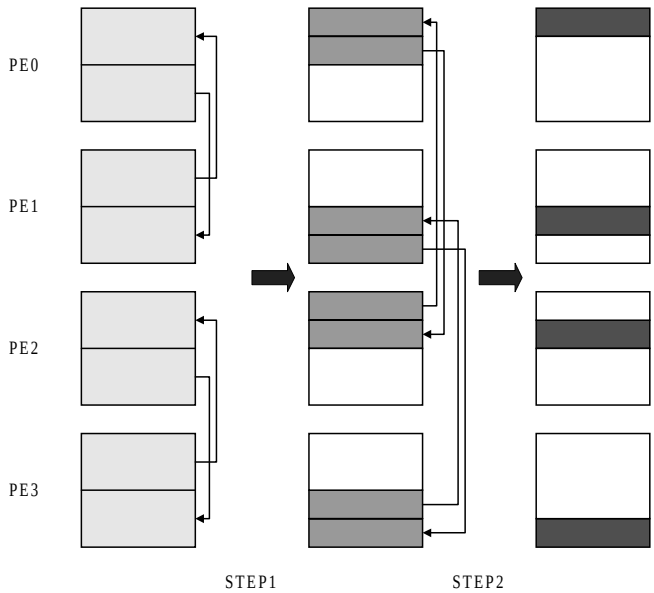

Figure 2. The binary-swap compositing method using four processors.

3.2. The binary-swap with bounding rectangle (BSBR) Method

When applying the bounding rectangle scheme to the binary-swap compositing method, termed BSBR, we have the following two cases. We show these two cases in Figure 4. In the first case shown in Figure 4(a), for each processor pair, PE and PE′, PE needs to send (receive) pixels to (from) PE′ if the sending (receiving) subimage contains a portion of the bounding rectangle. The portion of a bounding rectangle sent (received) to (from) PE′ is called the sending (receiving) bounding rectangle of PE. The portion of a bounding rectangle that is retained in PE is called the local bounding rectangle of PE. For the second case shown in Figure 4(b), for

---

Algorithm Binary_Swap_Compositing_Method(*P*)

1. for all processors do in parallel
2. **for** $i = 1$ to log$P$ do
3. Each PE sends the half subimage that it keeps to PE';
4. Each PE receives the half subimage from PE';
5. Each PE composites incoming half subimage with its local half subimage;
6. **endfor**

end_of_Binary_Swap_Compositing_Method

Figure 3. The algorithm of the binary-swap compositing method.

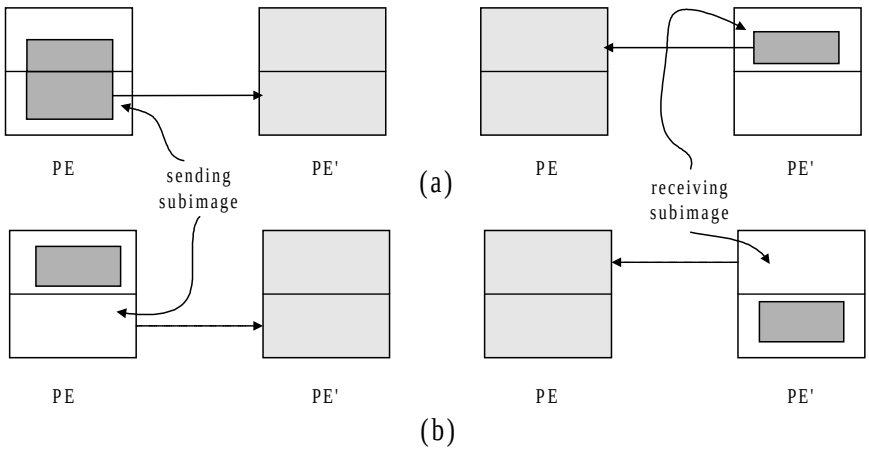

Figure 4. Two cases for the BSBR method. (a) The first case. (b) The second case.

each processor pair, PE and PE′, PE needs not send (receive) pixels to (from) PE′ if the sending (receiving) subimage contains no portion of the bounding rectangle. In order to obtain the bounding rectangle information, the processor in each pair has to exchange its bounding rectangle information in each compositing stage.

The advantage of the BSBR method is that it can quickly find an approximate number of non-blank pixels with less additional fields to record these pixels' positions. In the BSBR method, it takes an $O(A)$ amount of time to search the sending bounding rectangle and local bounding rectangle in the first compositing stage. In the later compositing stages, each processor generates a new local bounding rectangle by comparing the local bounding rectangle and the receiving bounding rectangle information. The time complexity is $O(1)$ in comparing the bounding rectangle. The disadvantage of the BSBR method is that it sends not only non-blank pixels, but also blank pixels within the sending bounding rectangle. If there is a high density of non-blank pixels in a sending bounding rectangle, the BSBR method performs well. Conversely, it performs poorly as the non-blank pixels' density of a bounding rectangle is sparse.

The BSBR method is implemented as follows. We use four short integers to represent the upper-left and the lower-right coordinates of a bounding rectangle. First, each processor finds the boundary of the sending bounding rectangle and packs pixels in the rectangle into a sending buffer. Then, for each PE in a processor pair, it sends the sending buffer to PE′ and accepts the receiving bounding rectangle from PE′. If the receiving bounding rectangle contains no pixels, the pixel compositing is completed in this compositing stage. Otherwise, the pixels in the receiving bounding rectangle are composited with the pixels in the local bounding rectangle. The compositing time of the BSBR method is $O(A^k_{rec})$ at the kth compositing stage, where A^k_{rec} is the number of pixels in a receiving bounding rectangle.

In the BSBR method, it requires $\log P$ communication stages to send and receive the bounding rectangle information for each paired processors. The number of empty bounding rectangles depends on the number of processors and the rotation

of a viewing point. As the number of processors increases, the ratio of empty bounding rectangle increases as well. The factors of a viewing point are rotation dimension and rotation degree. For each processor, there are $\log \sqrt[3]{P}$ nonempty bounding rectangles in the receiving bounding rectangles when we use a normal orthogonal projection. As a viewing point rotates along one axis, each processor has a maximum of $\log(\sqrt[3]{P^2})$ nonempty bounding rectangles in $\log P$ communication stages. Each processor has a maximum of $\log P$ nonempty bounding rectangles in $\log P$ communication stages while a viewing point rotates along two axes. For each processor, the local computation time and the communication time for the BSBR method are

$$T_{comp}(BSBR) = T_{bound} + \sum_{k=1}^{\log P} \left(T_o \times A_{rec}^k [B(k)] \right) \text{ and} \tag{3}$$

$$T_{comm}(BSBR) = \sum_{k=1}^{\log P} \left(T_s + \left(8 + 16 \cdot A_{rec}^k [B(k)] \right) \times T_c \right), \tag{4}$$

where T_{bound} is the computation time for finding a sending bounding rectangle and a local bounding rectangle in the first compositing stage, and

$$[B(k)] = \begin{cases} 1, & \text{if a receiving bounding rectangle is not empty} \\ 0, & \text{if a receiving bounding rectangle is empty} \end{cases}.$$

3.3. The binary-swap with run-length encoding and static load-balancing (BSLC) method

The run-length encoding is better than explicit x and y coordinates because it uses less position information to record non-blank pixels. In [1], they used the values of pixels to do encoding. It is a good scheme for surface rendering, but not efficient for volume rendering due to an additional field used to record a count of the pixels with the same value. In surface rendering or polygon rendering, a pixel's value is usually represented by an integer. Due to the data coherence of 2D images and the pixel's value representation format, the count field can be used efficiently. It can compress many pixels with the same value into one pixel. However, in volume rendering, opacity and intensity are used as a pixel's values and are usually represented by floating points. In general, the values of a non-blank pixel and the one next to it are different. If we apply the run-length encoding method used in [1] for volume rendering, the result image size is usually equal to the number of non-blank pixels of the original image. This will increase the message size due to the count field. To avoid this problem, we use the background/foreground value of a pixel (blank/non-blank) instead of the value of a pixel to do encoding.

Figure 5 shows the case of run-length encoding by using the background/foreground values of pixels. In the run-length encoding, transmitting non-blank pixels and compositing pixels may not be balanced because of uneven non-blank pixel distribution, which can be corrected by using static load-balancing methods. An interleaved array distribution is a good choice for balancing a compositing load without significant processor overheads. Figure 6 shows the load-balancing scheme of an interleaved array distribution in the binary-swap compositing method.

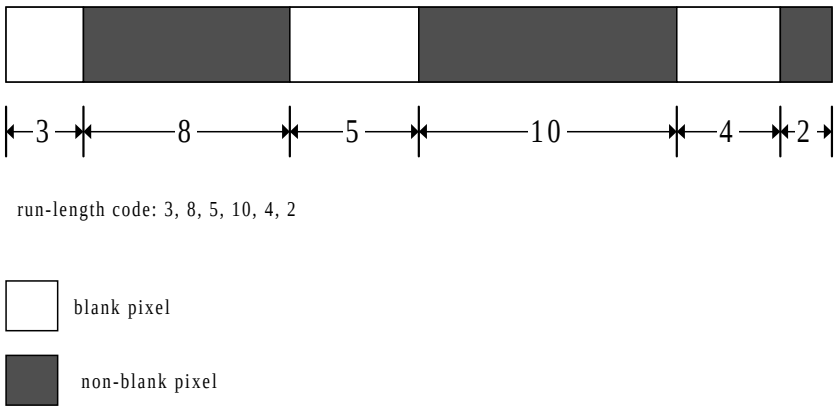

Figure 5. The case of run-length encoding by using the background/foreground values of pixels.

In the BSLC method, the rule of data exchange is the same as the binary-swap compositing method. The difference is that the half of a subimage we send is in interleaved sections instead of a whole block. In the run-length encoding, iterating through the pixels of the image using an interleaved method, the algorithm checks a pixel's value (opacity or intensity) to see whether it is zero or nonzero, i.e., the pixel is blank or non-blank. The algorithm records the numbers of the continuous

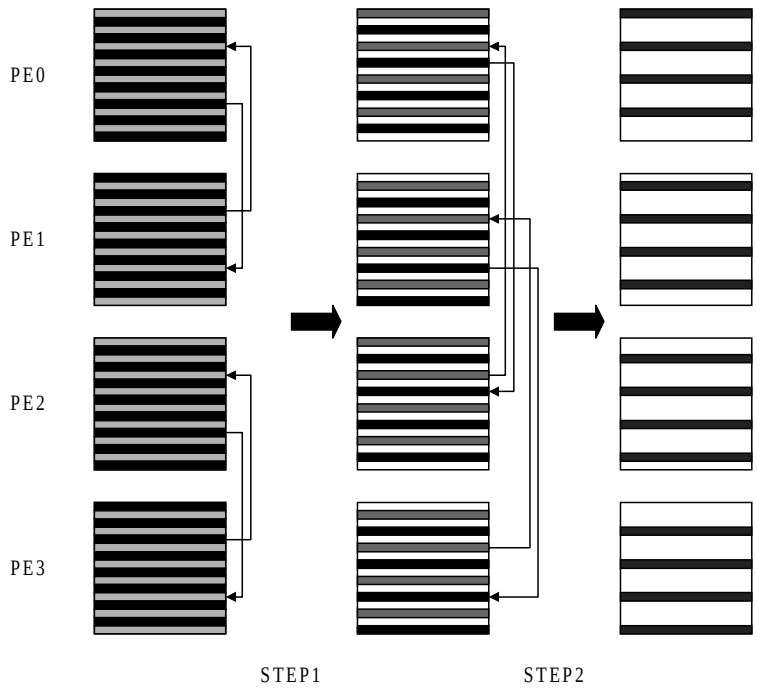

Figure 6. An interleaved array distribution scheme in the binary-swap compositing method.

blank and non-blank pixels, as shown in Figure 5. After encoding, we generate the run-length codes to index the blank and non-blank pixels. Then we pack the run-length codes and non-blank pixels into a sending buffer. As one processor receives the data from the other paired processor, it only composites the non-blank pixels in a receiving buffer according to the run-length codes.

The time complexity of a run-length encoding phase in the BSLC method is $O(A/2^k)$ at the kth compositing stage. The size of run-length codes depends on an image. As the image contains almost continuous blank and non-blank pixels, it generates fewer codes than blank and non-blank pixels in a discrete distribution. In the worst case, i.e., the blank and non-blank pixels appear in turn, the size of run-length codes is equal to the scheme of explicit x and y coordinates. The compositing time in the BSLC method is $O(A^k_{opaque})$ at the kth compositing stage, where A^k_{opaque} is the number of non-blank pixels in a receiving subimage. For each processor, the local computation time and the communication time in the BSLC method are

$$T_{comp}(BSLC) = \sum_{k=1}^{\log P} \left(T_{encode} \times \frac{A}{2^k} + T_o \times A^k_{opaque} \right) \text{ and} \tag{5}$$

$$T_{comm}(BSLC) = \sum_{k=1}^{\log P} \left(T_s + \left(2 \cdot R^k_{code} + 16 \cdot A^k_{opaque} \right) \times T_c \right), \tag{6}$$

where T_{encode} is the computation time of run-length encoding per pixel and R^k_{code} is the size of run-length codes. Each element of run-length codes is represented by two bytes.

3.4. The binary-swap with bounding rectangle and run-length encoding (BSBRC) method

The disadvantage of the BSLC method is that it has to iterate through all the pixels of a sending subimage whether the pixels are blank or non-blank. The disadvantage of the BSBR method is that if the bounding rectangle is sparse, the processor sends too many blank pixels to the paired processor. By combining the bounding rectangle and the run-length encoding, the disadvantages of the BSBR method and the BSLC method can be avoided. We call this new method BSBRC. In the BSBRC method, a processor not only requires less computing time by using the bounding rectangle but also sends less data to the paired processor with the run-length encoding. In the BSLC method, or the method proposed by Ahrens *et al.* [1], it has to iterate through all pixels in the sending subimage in the run-length encoding phase. The BSBRC method only iterates through the pixels in the sending bounding rectangle of the subimage. In the run-length encoding phase, a processor processes the pixels within the sending bounding rectangle. It reduces the encoding time and generates fewer run-length codes. In the compositing phase, a processor composites only the non-blank pixels instead of all the pixels in the receiving subimage according to the run-length encoding. It takes less compositing time and sends out less data since

the total number of the run-length codes and non-blank pixels is less than the number of pixels of an image. The BSBRC algorithm is given as follows.

Algorithm BSBRC(P) {

1. For all PEs do in parallel
2. /* Find the bounding rectangle*/
3. For all pixels in the subimage do
4. Find the boundary of the local bounding rectangle to cover all non-blank pixels;
5. For $k = 1$ to $\log P$ do {
6. Use the centerline of the subimage to divide the local bounding rectangle into the new local bounding rectangle and the sending bounding rectangle;
7. For all pixels in the boundary of the sending bounding rectangle do
8. Use the run-length encoding to generate the codes to index the non-blank pixels and pack non-blank pixels into a temporary buffer;
9. Pack the sending bounding rectangle information into the sending buffer;
10. If the sending bounding rectangle is not empty {
11. Pack the run-length codes into the sending buffer;
12. Pack the pixels in a temporary buffer into the sending buffer;
}
13. Send the sending buffer to the paired PE′;
14. Receive the receiving buffer from the paired PE′;
15. Unpack the receiving bounding rectangle information from the receiving buffer;
16. If the receiving bounding rectangle is not empty {
17. Unpack the run-length codes from the receiving buffer;
18. Unpack the pixels from the receiving buffer into a compositing buffer;
19. For each pixel in a compositing buffer do
20. Composite the pixel with the corresponding pixel in the local subimage according to the run-length codes
}
21. Calculate the new local bounding rectangle by combining the local bounding rectangle with the receiving bounding rectangle;
}

} *end_ of_ BSBRC*

For each processor, the local computation time and the communication time for the BSBRC method are

$$T_{comp}(BSBRC) = T_{bound} + \sum_{k=1}^{\log P} \left(T_{encode} \times A_{send}^{k} + T_o \times A_{opaque}^{k} \right) \tag{7}$$

and

$$T_{comm}(BSBRC) = \sum_{k=1}^{\log P} \left(T_s + \left(8 + 2 \cdot R_{code}^{k} + 16 \cdot A_{opaque}^{k} \right) \times T_c \right), \tag{8}$$

where A_{send}^{k} is the number of pixels in a sending bounding rectangle at the kth compositing stage.

4. Performance study and experimental results

To evaluate the performance of the proposed methods, we have implemented these methods on an IBM SP2 parallel machine [5] along with the binary-swap (BS) compositing method. The IBM SP2 parallel machine belongs to the National Center of High Performance Computing (NCHC) in Taiwan. This super-scalar architecture uses a CPU model of IBM RISC System/6000 POWER2 with a clock rate of 66.7 MHz. There are eighty IBM POWER2 nodes in the system, and each node has a 128KB 1st-level data cache, a 32KB 1st-level instruction cache, and 128MB of memory space. Each node is connected to a low-latency, high-bandwidth interconnection network called the High Performance Switch (HPS).

These proposed methods were written in C language with an MPI [13] message passing library. The test samples are *Engine_low* ($256 \times 256 \times 110$), *Engine_high* ($256 \times 256 \times 110$), *Head* ($256 \times 256 \times 113$), and *Cube* ($256 \times 256 \times 110$). The images of the test samples are shown in Figure 7. In the rendering phase, for each test sample, a ray tracing algorithm is used to generate 8-bit gray-level images in two

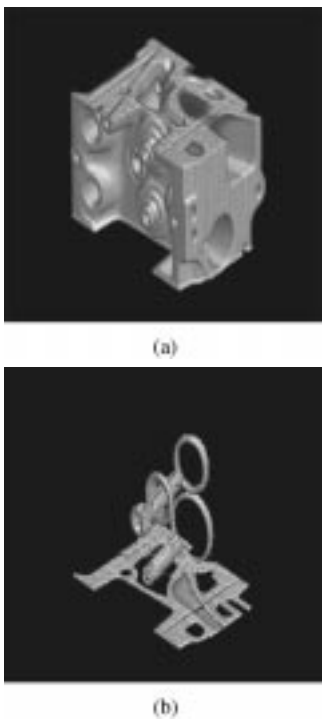

Figure 7. Test sample images. (a) *Engine_low*, (b) *Engine_high*, (c) *Head*, (d) *Cube*.

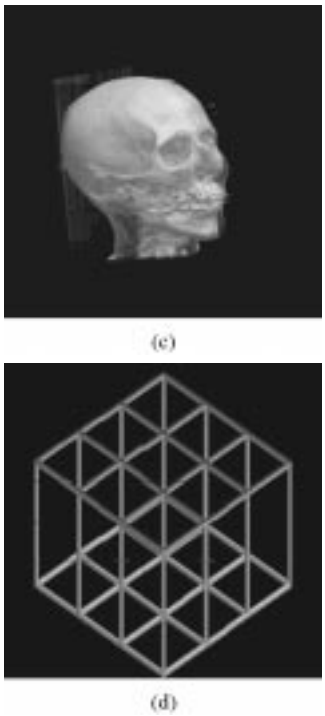

Figure 7. Continued.

different sizes, 384 × 384 pixels and 768 × 768 pixels. In our experiments, we run the test samples on 2, 4, 8, 16, 32, and 64 processors.

We use the maximum received message size to evaluate the performance of the proposed methods. For each processor, it calculates the message sizes it received at all compositing stages by $m_i = \sum_{k=1}^{\log P}(R_i^k)$, where R_i^k is the received message size in bytes for the ith processor at the kth compositing stage. The maximum received message size, $M_{\max}$, among P processors is defined as $M_{\max} = MAX_{i=0}^{P-1}(m_i)$. From Equations (2), (4), (6), and (8), in general, we have that

$$M_{\max}^{BS} \geq M_{\max}^{BSBR} \geq M_{\max}^{BSBRC} \geq M_{\max}^{BSLC} \tag{9}$$

where $M_{\max}^{BS}$, $M_{\max}^{BSBR}$, $M_{\max}^{BSBRC}$, and $M_{\max}^{BSLC}$ are $M_{\max}$ of the BS, BSBR, BSBRC, and BSLC methods, respectively.

Table 1 shows the compositing time ($T_{total}(L) = T_{comp}(L) + T_{comm}(L)$) for each proposed method L and the BS method for the test images with 384 × 384 pixels. From Table 1, we can see that in most cases the BS method has the largest communication time and the BSLC method has the smallest communication time while $T_{comm}(BSBRC)$ is less than $T_{comm}(BSBR)$. For the sparser cases, *Engine_high* and *Cube*, $T_{comm}(BSLC)$ is much less than $T_{comm}(BSBR)$ and $T_{comm}(BSBRC)$. In a few cases where the number of processors is two, $T_{comm}(BSLC)$ is larger than $T_{comm}(BSBRC)$. The reason is that $A_{opaque}^k(BSLC)$ is almost equal to $A_{opaque}^k(BSBRC)$, but the BSLC method has more run-length code than the BSBRC

Table 1. The compositing time of proposed methods for the four 384 × 384 test images

Number of processors	BS			BSBR			BSLC			BSBRC		
	T_{comp}	T_{comm}	T_{total}	T_{comp}	T_{comm}	T_{total}	T_{comp}	T_{comm}	T_{total}	T_{comp}	T_{comm}	T_{total}
						Engine_low						
2	297.85	29.25	327.10	70.72	11.57	82.29	85.01	9.97	94.98	64.20	8.72	72.92
4	369.18	55.34	424.52	60.68	30.92	91.60	119.29	17.10	136.39	60.97	28.01	88.98
8	371.34	56.11	427.44	57.37	44.41	101.78	131.91	11.67	143.59	69.31	29.80	99.11
16	378.98	59.01	438.00	56.26	44.25	100.51	136.67	16.70	153.36	60.13	30.27	90.40
32	385.80	60.56	446.36	48.78	43.19	91.97	141.50	8.51	150.02	60.82	22.41	83.23
64	392.48	60.32	452.80	50.72	39.99	90.72	140.75	11.09	151.85	59.12	24.56	83.68
						Engine_high						
2	298.03	26.98	325.02	59.88	21.95	81.83	71.48	2.83	74.30	57.97	5.52	63.49
4	371.75	57.24	428.99	54.26	30.28	84.54	107.07	3.87	110.94	52.88	13.66	66.54
8	371.42	64.38	435.81	54.00	37.75	91.75	123.27	4.84	128.11	55.39	23.05	78.44
16	373.97	66.54	440.51	52.07	30.40	82.46	132.18	5.87	138.05	50.02	23.10	73.12
32	388.80	64.15	452.95	42.70	39.28	81.99	134.47	4.28	138.76	48.35	25.75	74.10
64	392.95	62.22	455.17	48.04	26.19	74.24	137.94	5.41	143.35	48.01	20.25	68.27
						Head						
2	290.90	35.64	326.53	73.62	20.90	94.52	88.51	10.43	98.94	65.56	9.72	75.28
4	364.66	50.25	414.90	71.00	42.78	113.78	127.21	16.96	144.17	63.21	34.80	98.02
8	389.83	59.83	449.66	62.79	42.43	105.22	135.47	14.38	149.85	73.46	32.88	106.34
16	388.89	60.66	449.55	51.23	37.75	88.98	142.15	15.46	157.61	60.11	34.74	94.85
32	382.29	59.96	442.58	51.16	45.01	96.17	137.41	15.57	152.99	54.97	42.66	97.63
64	385.96	62.97	448.93	52.74	33.51	86.26	138.25	11.47	149.73	64.35	17.57	81.93
						Cube						
2	299.04	27.12	326.15	81.86	14.03	95.89	73.21	4.70	77.91	64.26	3.30	67.56
4	374.49	55.07	429.56	69.78	44.38	114.16	108.52	6.32	114.84	59.27	17.91	77.18
8	373.64	63.52	437.16	75.21	57.74	132.95	124.28	7.62	131.90	69.59	17.25	86.83
16	380.22	60.38	440.60	59.31	50.83	110.13	131.59	7.98	139.57	55.78	26.76	82.54
32	381.10	61.48	442.58	47.16	42.15	89.31	139.81	4.53	144.34	51.97	19.46	71.44
64	383.94	64.15	448.09	47.66	27.05	74.72	137.34	9.33	146.68	54.70	11.33	66.03

(Time unit: ms).

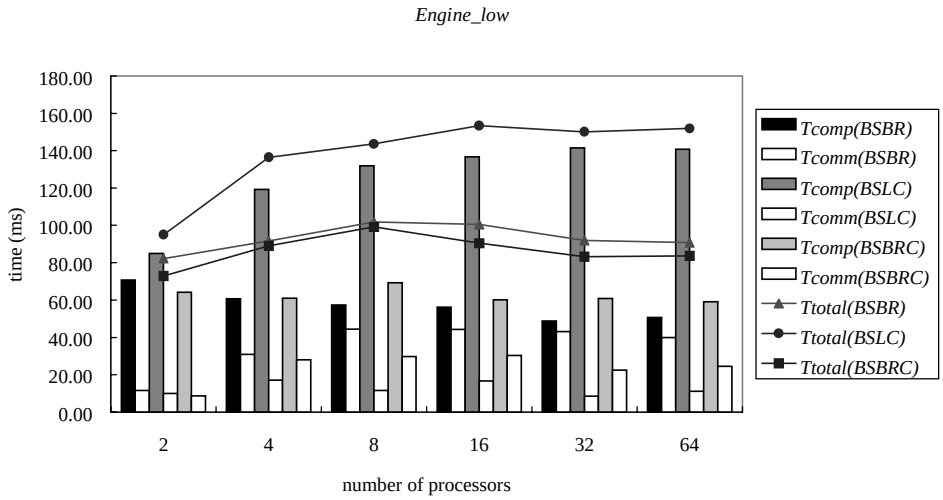

Figure 8. The compositing time of the BSBR, BSLC, and BSBRC methods for *Engine_low*.

method. The experiment results of the communication time in Table 1 match the analysis of Equation (9).

For the computation time, $T_{comp}(BSLC)$ is much larger than $T_{comp}(BSBRC)$ and $T_{comp}(BSBR)$. From Equation (1) and Equation (5), we can predict an asymptotic bound for the computation time of the BS method and the BSLC method respectively, when the number of processors exceeds a threshold.

To visualize the experiment results better, we use Figures 8 and 9 to show the compositing time of the three proposed methods for *Engine_low* and *Head*, respectively. The images of these two test samples are denser than the other two. Later

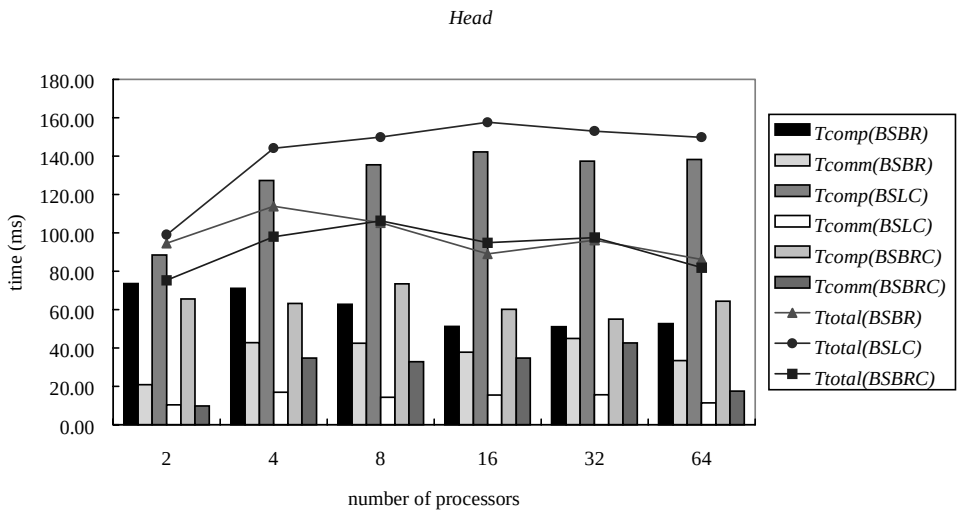

Figure 9. The compositing time of the BSBR, BSLC, and BSBRC methods for *Head*.

we will show the figures for *Engine_high* and *Cube* that have better results as was expected.

In the experiments where the number of processors is greater than eight, the $T_{comp}(BSBRC)$ of some cases is larger than the $T_{comp}(BSBR)$. However, in Figure 8, every $T_{total}(BSBRC)$ is less than $T_{total}(BSBR)$ because the difference between $T_{comm}(BSBRC)$ and $T_{comm}(BSBR)$ is much larger than the difference between $T_{comp}(BSBRC)$ and $T_{comp}(BSBR)$. Although $T_{total}(BSBRC)$ is larger than $T_{total}(BSBR)$ in Figure 9 when the number of processors is 8, 16, or 32, the difference is very small. $T_{total}(BSLC)$ is much larger than $T_{total}(BSBR)$ as well as $T_{total}(BSBRC)$ due to a large $T_{comp}(BSLC)$, even though $T_{comm}(BSLC)$ is the smallest among the four methods.

Figures 10 and 11 show the compositing time of the three proposed methods for *Engine_high* and *Cube*, respectively. In these cases of sparser images, $T_{comp}(BSBRC)$ is larger than $T_{comp}(BSBR)$ when the number of processors is greater than thirty-two. However, the BSBRC method performs much better than other methods, especially when the bounding rectangle of a subimage is larger and sparser, such as *Cube*. In Figure 11, $T_{total}(BSBRC)$ is much less than $T_{total}(BSBR)$ in all test cases. Also note that $T_{total}(BSLC)$ is less than $T_{total}(BSBR)$ only when the number of processors is less than eight.

Table 2 shows the compositing time of the three proposed methods for the test samples with 768×768 pixels. The results are similar to those of Table 1. In general, the BSBRC method has the best overall performance among them. A further analysis will be given below.

The goal of our proposed methods is to reduce the compositing time ($T_{total}(L) = T_{comp}(L) + T_{comm}(L)$). From Equation (9) and our experiments, we know that $T_{comm}(BSLC)$ is the smallest, but $T_{comp}(BSLC)$ is the largest among the four methods. Since $T_{comp}(BSLC)$ dominates the total time due to the encoding of each

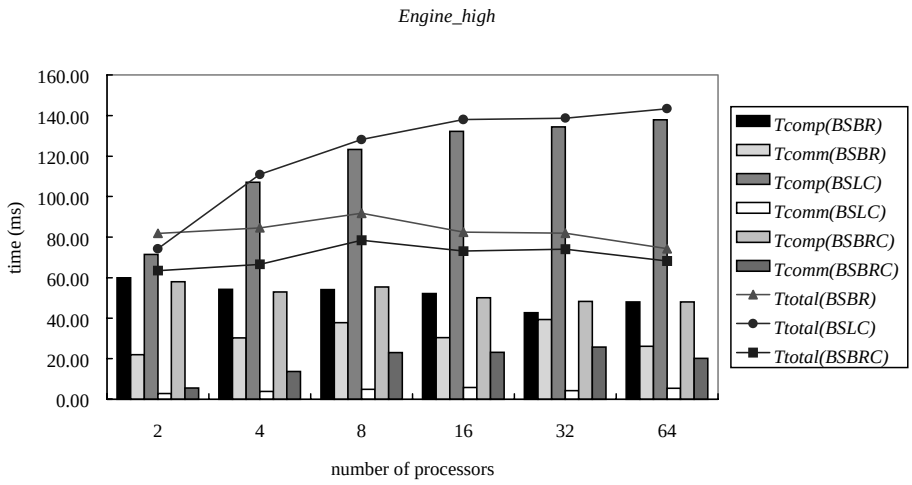

Figure 10. The compositing time of the BSBR, BSLC, and BSBRC methods for *Engine_high*.

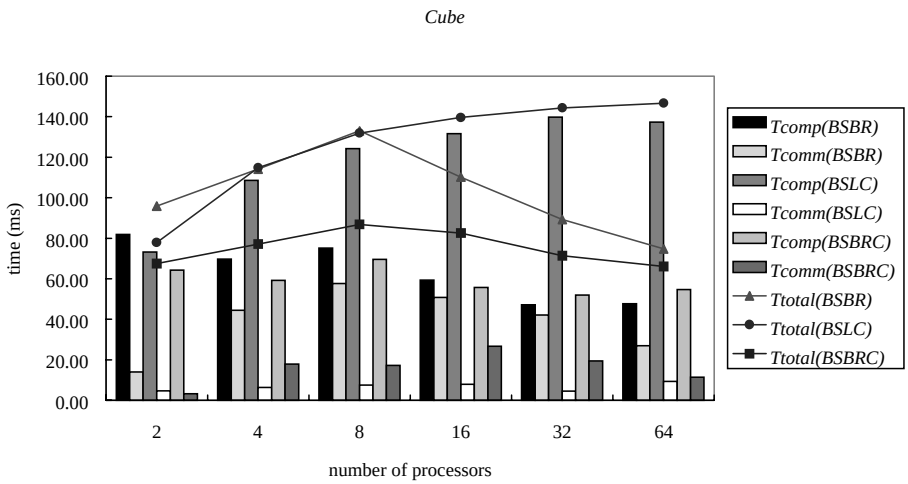

Figure 11. The compositing time of the BSBR, BSLC, and BSBRC methods for *Cube*.

image ($A/2^k$ from Equation (5)) to achieve the load balancing, the BSLC method is not as good as the other two.

From Equations (4) and (8), we get that $T_{comm}(BSBRC)$ is less than $T_{comm}(BSBR)$ except when the pixels in a receiving bounding rectangle are all non-blank. The difference between $T_{comm}(BSBRC)$ and $T_{comm}(BSBR)$ becomes larger when a receiving bounding rectangle is sparser. Although in some cases $T_{comp}(BSBRC)$ is larger than $T_{comp}(BSBR)$ depending on the factors of T_{encode}, T_o, A^k_{send}, and A^k_{opaque}, the total compositing time of the BSBRC method is less than the BSBR method, i.e., $(T_{comm}(BSBR) - T_{comm}(BSBRC)) > (T_{comp}(BSBRC) - (T_{comm}(BSBR))$.

The overall efficiency of the BSBRC method is better than the BSBR and BSLC methods, but it also has bottlenecks. First, the BSBRC method requires extra computation time to encode the non-blank pixels. After encoding, it generates additional codes to index non-blank pixels thus increasing the communication time. Second, as the bounding rectangle becomes denser, the performance of the BSBR method is closer to the BSBRC method. Third, as the number of processors increases, the size of the subimage decreases such that the difference between $T_{total}(BSBRC)$ and $T_{total}(BSBR)$ gets smaller. Further efforts to improve our methods will be discussed in the next section.

5. Conclusions and future work

In this paper we have presented three compositing methods, the binary-swap with bounding rectangle (BSBR) method, the binary-swap with run-length encoding and static load-balancing (BSLC) method, and the binary-swap with bounding rectangle and run-length encoding (BSBRC) method, for the sort-last-sparse parallel volume rendering system and have demonstrated the performance improvements on a distributed memory multicomputer.

Table 2. The compositing time of proposed methods for the four 768 × 768 test samples

Number of processors	BSBR			BSLC			BSBRC		
	T_{comp}	T_{comm}	T_{total}	T_{comp}	T_{comm}	T_{total}	T_{comp}	T_{comm}	T_{total}
					Engine_low				
2	343.75	60.66	404.41	336.63	30.38	367.00	314.63	44.06	358.68
4	323.38	151.64	475.02	480.83	53.26	534.09	322.16	117.12	439.28
8	370.87	193.27	564.14	578.53	43.01	621.55	349.70	138.78	488.48
16	290.41	195.00	485.41	554.43	76.93	631.37	248.74	154.76	403.51
32	213.03	238.19	451.22	568.99	40.51	609.50	272.68	125.78	398.46
64	231.22	162.73	393.95	578.29	25.42	603.72	270.51	109.17	379.68
					Engine_high				
2	305.77	79.38	385.15	288.96	9.69	298.65	269.64	42.86	312.50
4	316.46	154.02	470.48	430.95	12.72	443.67	247.90	60.01	307.91
8	323.85	160.80	484.65	490.93	19.14	510.07	234.16	119.68	353.84
16	219.62	150.89	370.51	517.86	29.39	547.25	259.29	35.43	294.72
32	191.24	237.84	429.08	535.04	19.55	554.59	202.16	129.53	331.70
64	193.53	124.72	318.25	542.41	12.75	555.16	206.82	57.40	264.23
					Head				
2	401.25	247.53	648.78	349.41	37.27	386.68	326.74	165.58	492.32
4	470.61	199.23	669.84	493.69	64.70	558.39	390.48	158.02	548.50
8	375.13	183.51	558.64	539.47	62.80	602.27	333.66	194.47	528.14
16	308.10	190.95	499.05	560.23	61.30	621.53	294.90	178.62	473.52
32	187.68	222.46	410.14	608.87	20.38	629.26	234.19	157.53	391.72
64	253.06	181.00	434.06	561.54	40.30	601.85	237.53	99.07	336.60
					Cube				
2	419.80	61.56	481.36	301.48	14.70	316.18	292.14	24.50	316.64
4	376.53	216.55	593.08	443.85	25.52	469.37	273.59	103.13	376.72
8	454.21	196.93	651.13	498.76	23.77	522.54	298.69	87.08	385.78
16	247.90	294.01	541.91	524.00	23.17	547.17	236.72	106.54	343.26
32	264.53	168.79	433.32	542.34	14.81	557.15	220.23	56.13	276.36
64	218.93	120.05	338.98	547.54	12.48	560.02	211.55	47.41	258.96

(Time unit: ms).

We have implemented these three methods along with the binary-swap method on an IBM SP2 parallel machine. From the experimental results, we found that the BSLC method has the smallest maximum received message size among the four methods. However, it requires more computation time than the BSBRC method. The BSBRC method has fewer maximum received message sizes than the BSBR method, and it also has the shortest compositing time among the four methods.

In the future, we first plan to improve the binary-swap compositing method running on any number of processors. The binary-swap compositing method is an efficient method for parallel rendering with more parallelism. The drawback of the binary-swap compositing method is that the number of processors must be a power of two. Second, we plan to implement the parallel splatting volume rendering [15] method and explore an efficient load-balancing scheme in the rendering phase since load-balancing plays an important role in sort-last parallel rendering as the size of opaque voxels has large disparities. Third, we will try our methods on different types of machine architectures, such as [16–18] and study more efficient encoding schemes.

Acknowledgments

The authors would like to thank the referees for their helpful comments. This research was partially supported by the National Science Council of Republic of China under contract NSC-89-2213-E-035-032.

References

1. J. Ahrens and J. Painter. Efficient sort-last rendering using compression-based image compositing. In *Proceedings of the 2nd Eurographics Workshop on Parallel Graphics & Visualization*, 1998.
2. M. Cox and P. Hanrahan. Pixel merging for object-parallel rendering: A distributed snooping algorithm. In *Proceedings of the 1993 Parallel Rendering Symposium*, pp. 49–56, New York, 1993.
3. J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice Second Edition in C.* Addison-Wesley, Reading, MA, 1990.
4. W.M. Hsu. Segmented ray casting for data parallel volume rendering. In *Proceedings of the 1993 Parallel Rendering Symposium*, pp. 7–14, San Jose, CA, October 1993.
5. IBM. IBM AIX Parallel Environment, Parallel Programming Subroutune Reference.
6. G. Johnson and J. Genetti. Volume rendering of large datasets on the Cray T3D. In *1996 Spring Proceedings (Cray User Group),* pp. 155–159, 1996.
7. P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Computer Graphics and Application*, 2:218–231, 1996.
8. T.Y. Lee, C.S. Raghavendra, and J.B. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics,* 2:202–217, 1996.
9. M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics,* 9:245–261, 1990.
10. W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21:163–169, 1987.
11. K.L. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Application,* 14:59–67, 1994.
12. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Application,* 14:23–32, 1994.

13. MPI Fourm. MPI: A message-passing interface standard, May 1994.
14. U. Neumann. Volume reconstruction and parallel rendering algorithms: A comparative analysis. Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, 1993.
15. L.A. Westover. SPLATTING: A parallel, feed-forward volume rendering algorithm. Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, July 1991.
16. M. Berekovic and P. Pirsch. An array processor with parallel data cache for image rendering and compositing. In *Proceedings of Computer Graphics International CGI'98,* pp. 411–414, June 1998.
17. B. Liu, M. Margala, N. Durdle, and S. Juskiwm. High-speed image composition with enhanced multiplier structures. In *Proceedings of the 1999 IEEE Canadian Conference on Electrical and Computer Engineering,* pp. 477–485, May 1999.
18. Kwan-Liu Ma. Parallel rendering of 3D AMR data on the SGI/Cray T3E. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation,* pp. 138–145, 1999.