

Parallel Volume Rendering on a Network of Workstations

Christopher Giertsen and
Johnny Petersen
IBM Bergen Environmental Sciences
and Solutions Centre

Parallel processing can address the often immense need for computing power in volume rendering. We present an efficient parallel algorithm where each processor computes sequences of lines in the picture.



Volume rendering can easily become so computation intensive that response times prohibit its practical use. There are cases, of course, where it is possible to achieve good response times; but for large data sets, large images, and animations, the need for computing power is immense. Several authors have commented on the use of parallel processing to reduce this problem,¹ but the literature includes very few parallel algorithms and performance results.

Approaches to parallel volume rendering can be grouped into three categories:

- Hardware architectures designed specifically for parallel volume rendering.²
- Software implementations on machines with hardware support for parallel volume rendering.³
- Parallel volume rendering algorithms implemented entirely in software on general-purpose hardware for parallel processing.^{4,5}

All these methods address volumes defined either by rectilinear node-valued elements or by voxels (that is, quantum-unit volume elements).

We present here the first algorithm for parallel volume rendering on general-purpose workstations connected to a local area network (LAN). The idea is to take advantage of the parallel processing capabilities inherent in such computing envi-

ronments. By using public domain software such as Parallel Virtual Machine (PVM),⁶ no additional investments are required.

The proposed algorithm is based on an efficient scan-line algorithm for volume rendering of irregular meshes.⁷ This algorithm computes images by intersecting the mesh with successive planes defined through each scan line and perpendicular to the screen. We call these planes *scan planes*. Image coherency from one scan plane to the next, and within each scan plane, speeds up image computation. The proposed algorithm is a modified version of the scan-line algorithm, suitable for parallelization and for handling large data sets efficiently. Based on an efficiency analysis of this version, we conclude that minimal additional computing and communication are required if each processor is given the task of computing sequences of successive lines in the image, hereafter called *sections*. We also suggest how to achieve good load balancing on a group of heterogeneous workstations that have arbitrary loads by other users.

Scan-line volume rendering

Previous methods for volume rendering of irregular meshes are based on ray casting⁸ or direct projection.⁹ These methods depend on either a 3D point-location algorithm or an algorithm for visibility ordering of meshed polyhedra. To our knowledge, no such algorithms can handle general meshes where some elements can be missing or inactive and where neighboring elements are not necessarily aligned at shared element faces.

Figure 1. Scan-line processing of an irregular volume element.

On the other hand, Giertsen's scan-line algorithm⁷ can handle holes, cavities, and displaced elements efficiently. It stores the computational mesh in an array, *elm*, where each entry is a complete definition of a hexahedral element, having eight vertices at which scalar values are available. Each volume element can then be manipulated as an independent entity.

The first step in the algorithm multiplies the coordinates of each volume element with a viewing transformation matrix, *tm*. Then the elements are sorted using the highest *y* value in each element as the key. The purpose is to avoid searching through all elements for each scan line to determine the active elements, that is, those that contribute to the pixel values.

Next, an image is drawn line by line. For each line, the algorithm determines the interval of active elements in *elm* by a procedure that returns two pointers—called *first* and *last*—into *elm*. The algorithm then treats each active element independently in several steps. In the first step, it determines intersections between the element edges and the scan plane by the procedure *Slice()* and returns them in an array *i* (Figure 1a). Then, the procedure *PolyConstruct()* constructs one or more polygons from the points in *i* and returns the result in *p*, an array of polygons.

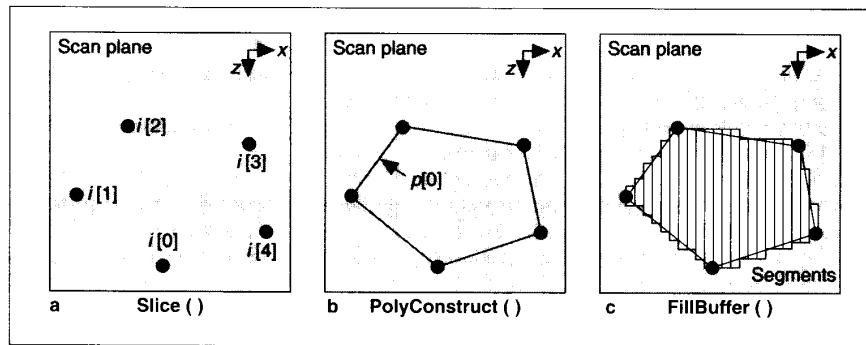
The simplest implementation of *PolyConstruct()* returns the polygon of intersection between the element and the scan plane (Figure 1b). However, this approach makes the interpolation of scalar values within the elements view-dependent. We can avoid this problem by subdividing the polygon of intersection.⁷

The final step is to scan-convert each polygon in *p* in increasing order of *x*. For each value of *x*, the span between the front and the back edges of the polygon in *p* constitutes a *segment*, the smallest entity in the scan-line volume rendering algorithm (Figure 1c). The algorithm computes a color and brightness for each segment.^{7,10} It stores these values in the scan-plane buffer, denoted *sb*, a 2D array used to order all visible contributions to the scan-line pixel values from all the active elements. *FillBuffer()* determines segments incrementally and performs insertion in *sb*. The scan-line pixels are then computed by traversing *sb* and by compositing contributions⁷ from different segments. The algorithm uses *AddContributions()* for this purpose and returns the result in an array called *sl*. This array may be copied to the screen or stored in an image buffer.

The pseudocode in Figure 2 shows the modified scan-line algorithm. Note that input parameters to the procedures are indicated by lowercase and the return parameters by uppercase letters.

Message passing

Our approach to parallel computing uses message passing on a number of heterogeneous workstations. Each workstation has its own private memory and is connected to a LAN, such as Token Ring or Ethernet. Message passing in this context means,



```

TransformElements(tm,ne,ELM);
SortElements(ne,ELM);
for (all scan lines y) {
  ActiveElements(y,FIRST,LAST);
  for (j=first; j<=last; ++j) {
    Slice(elm[j],y,I,NI);
    PolyConstruct(i,ni,P,NP);
    for (k=0; k<=np-1; ++k)
      FillBuffer(p[k],dm,SB);
  }
  for (all points x on the scan line)
    AddContributions(sb,dm,x,SL);
}

```

Variables used in the algorithm:

<i>tm</i>	: view transformation matrix
<i>elm</i>	: array of volume elements
<i>ne</i>	: number of elements
<i>y</i>	: current scan line
<i>first,last</i>	: pointers defining active elements
<i>i</i>	: intersection points
<i>ni</i>	: number of intersections in <i>i</i>
<i>p</i>	: array of polygons constructed from <i>i</i>
<i>np</i>	: number of polygons in <i>p</i>
<i>dm</i>	: rendering parameters
<i>sb</i>	: scan plane buffer
<i>x</i>	: x coordinate
<i>sl</i>	: array of scan line pixels
<i>j,k</i>	: loop variables

Figure 2. Serial volume rendering algorithm. Lowercase variable letters indicate input parameters; uppercase letters indicate return parameters.

first, that the program includes functions to transmit program data between the workstations during execution, and second, that a message passing daemon runs in the background on all of the workstations to synchronize the communication. This setup associates a unique identifier with each parallel program, for example, a process number.

We use a master-slave computing scheme. The master sends out jobs to the slaves; each slave finishes the job and sends the result back to the master, who sends out another job if any remain. In our terminology, we refer to the workstations where the master and the slaves are running as host and nodes, respectively.

Efficiency analysis

Analyzing the complexities of the scan-line algorithm provides useful information on how to construct the parallel algorithm and performance tests.

The average time complexity is difficult to derive because some factors that affect response time do not have a natural average case. An example is the view orientation. However, we can derive the worst-case time complexity, then draw some conclusions about the average case from that result.

The viewing transformation is a loop through all the elements, a process of $O(n)$, where n is the number of elements. By using Quicksort in `SortElements()`, the worst-case time complexity for this procedure is $O(n^2)$. The complexity of the initialization prior to scan-line processing is thus of $O(n^2)$.

To derive an expression for the main body, it is natural to start by considering the outer loop. If the number of lines in the screen window covered by the volume is denoted w_y , all operations within the loop on y are repeated w_y times. In the worst case, all volume elements cover all these scan lines, implying that for each line, `Slice()` and `PolyConstruct()` are called n times. In the following, the constants k_s and k_p express the maximum time spent in each call to these two procedures, respectively. The innermost loop is also entered n times, and in each execution, the number of calls to `FillBuffer()` is bounded by the maximum possible number of polygons in the array p , denoted p_m . In the worst case, the x extent of each polygon is identical to the maximum extent in the x direction, denoted w_x . If k_f is the maximum time required to compute one segment, the total work in `FillBuffer()` on one polygon can be expressed as $k_f w_x$.

After the scan plane buffer is filled with contributions from all active elements, `AddContributions()` can, in the worst case, be called w_x times. If the constant k_a represents the maximum time required to add all contributions for a pixel, the total time spent in the main body can be expressed as

$$w_y(n(k_s + k_p + p_m k_f w_x) + k_a w_x)$$

The constants k_s and k_p are small compared to $p_m k_f w_x$ and we can ignore them in the complexity analysis. Then the

expression simplifies to

$$w_y(w_x(p_m k_f n + k_a))$$

Again, k_a is small compared with $n p_m k_f$ and can be ignored. $p_m k_f$ is a constant, implying that the worst-case complexity of the main body equals $O(w_y w_x n)$. We cannot treat w_x and w_y as constants and conclude that the main body is of $O(n)$ because the factor $w_x w_y$ is, for irregular data sets, usually several orders of magnitude larger than n . Added with the time for initialization, the total worst-case complexity T_w is

$$T_w(n, w_x, w_y) = O(n^2 + w_y w_x n)$$

For large images, both w_y and w_x are on the order of 1,000. n is usually of less than 20,000 in our data sets, implying that the initialization counts for a small portion of the computations compared to the scan-line processing. We believe this is also true for the average case, where the initialization is still dominated by Quicksort but reduced to $O(n \log_2 n)$. In this case, the assumption that there are n active elements at each scan line is unrealistic. If we assume the elements to be evenly distributed in space, for example, as a cubic mesh, $n^{2/3}$ number of active elements becomes a more realistic estimate. Then the average time complexity T_a can be expressed as:

$$T_a(n, w_x, w_y) = O(n \log_2 n + w_x w_y n^{2/3})$$

Our conclusions regarding the first term in the expressions for T_w and T_a are important for parallel execution on a network of workstations. On a low-bandwidth LAN, we achieve high performance by computing the viewing transformations and the sorting for the whole data set on each node. The alternative is to perform these two operations once in the host program and then transfer to the nodes only the data necessary to produce sections of the image. This solution introduces a heavy load on the network and causes synchronization problems.

We need general message-passing functions to describe the flow of data between the host and the nodes. Two functions—`Send(pnum, memloc, numbytes)` and `Receive(pnum, memloc, numbytes)`—serve this purpose. The first sends data to a parallel process defined by the integer `pnum`, and the latter receives data from a parallel process defined by `pnum`. In both function calls, the variable `memloc` defines a start address for the message data, and the variable `numbytes` defines the number of bytes in the message. In practice, message-passing functions also include a type identifier so that processes receiving messages can select a particular packet of information.⁶ We chose to simplify the pseudocode by excluding the type identifier.

Processing on the host

There are at least two good reasons for using a domain decomposition where each processor is responsible for computing one or more sections. One is that scan-line volume rendering has

proved fast and suitable for handling most types of meshes. The other reason is that algorithm complexity is not affected (see the sidebar, "Efficiency analysis"). In the worst case, initializing a section involves a loop through all elements.

The key issue then is to achieve a good load balance, ensuring that all processors finish at the same time and none spend much time waiting idly on the others. Since we wish to achieve high parallel algorithm performance in a realistic environment, we have to consider that the network may consist of workstations with different processing capabilities and that any workstation may at any time be loaded by different users. These constraints indicate that we should use far more sections than the number of available processors. If we use a small number of sections, it is unlikely that the processing on each node will complete at the same time.

We suggest the following strategy for load balancing: The host initially gives each node processor the task of processing

```

/* broadcast the data set */
for (j=0; j<nprocs; ++j)
  Send(j,elm,sizeof(struct elm));
for (each new image to be computed) {
  /* broadcast the image parameters */
  Read(TM,DM);
  for (j=0; j<nprocs; ++j) {
    Send(j,tm,sizeof(struct tm));
    Send(j,dm,sizeof(struct dm));
  }
  /* send one section to each node processor */
  for (j=0; j<nprocs; ++j) {
    from=wy(nsect-j)/nsect-1;
    to=wy(nsect-1-j)/nsect;
    Send(j,from,4);
    Send(j,to,4);
  }
  for (j=nprocs; j<nsect; ++j) {
    /* receive and draw pixels... */
    Receive(PNUM,FROM,4);
    Receive(PNUM,TO,4);
    Receive(PNUM,BITMAP,4(from-to+1)wx);
    DrawRectangle(bitmap,from,to);
    /* ...and send a new section to pnum */
    from=wy(nsect-j)/nsect-1;
    to=wy(nsect-1-j)/nsect;
    Send(pnum,from,4);
    Send(pnum,to,4);
  }
  /* receive and draw the last nprocs sections */
  for (j=0; j<nprocs; ++j) {
    Receive(PNUM,FROM,4);
    Receive(PNUM,TO,4);
    Receive(PNUM,BITMAP,4(from-to+1)wx);
    DrawRectangle(bitmap,from,to);
  }
}

```

New variables:

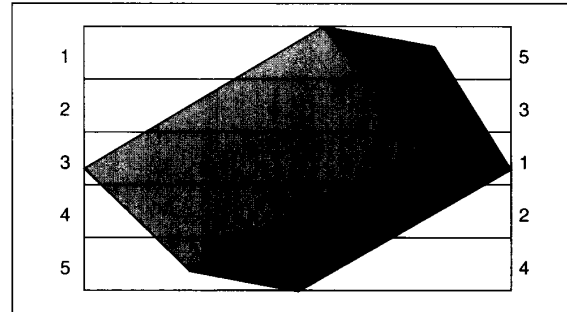
nprocs : the number of processors
nsect : the number of sections
from : the start of a section
to : the last line of a section
bitmap : a rectangle of pixels

one section, that is, of computing a rectangle in the final image. The host then waits for the first rectangle of pixels. When it receives a rectangle, it displays it immediately and sends a new section to that processor. This continues until there are no more sections to distribute. In the final step, the host waits for the last rectangles and draws them in the order received.

The pseudocode in Figure 3 is designed according to the proposed strategy. However, we also show how the data is first sent to each node and where to input the image parameters. This demonstrates how the parallel algorithm can work efficiently, for example, when a user wishes to generate a sequence of pictures with different view orientations and rendering parameters. The algorithm is written in the style of the C language and uses the procedure DrawRectangle() to draw on screen the rectangle of pixels received from a processor. No error han-

Figure 3. The algorithm on the host.

Figure 4. The order of the load distribution.



dling and loop termination criteria are included. Neither do we consider the initialization of the parallel processes and graphics hardware to be important in this context. We assume that both a pixel and an integer are represented by four bytes.

Our load-balancing algorithm makes no conceptual distinction between a busy high-performance processor and a low-performance processor. Each processor must deliver results before it gets more work. Thus, our algorithm is well suited for a group of heterogeneous workstations. Since a large number of sections are needed to ensure that all processors finish at roughly the same time, the initialization of each section is critical. In our discussion of parallel algorithm performance, we show that initialization has almost no impact on total computing time.

The algorithm in Figure 3 distributes sections to the available processors in sequential order, as illustrated by the numbers on the left side of Figure 4. We can improve performance by taking advantage of the concentration of rendering work in the middle sections.

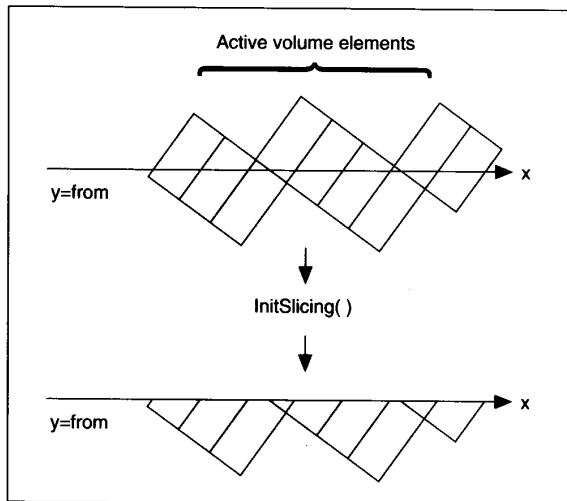
By starting from the middle, as illustrated by the numbers on the right side of Figure 4, the computation-intensive sections will be distributed first. As the rendering progresses, the work needed to process each section decreases. If a low-performance processor is given the task of computing the last section, the total image will still be completed quickly, since the amount of work is small. Thus, starting from the middle helps to ensure that all processors finish at almost the same time. The algorithm in Figure 3 introduces this change by replacing the variable j in the expressions for the variables $from$ and to :

$$\begin{aligned} & nsect/2 - (1 + j)/2 \quad \text{when } j \text{ is even} \\ & nsect/2 + (1 + j)/2 \quad \text{when } j \text{ is odd} \end{aligned}$$

We used this method in the performance tests described later in this article.

Processing on each node

The algorithm running on each node processor consists of two main parts. The first is an outer body containing the counterparts of the message-passing calls in the host process. The second is an inner loop for processing sections based on the scan-line volume rendering algorithm. In this loop we introduce two new functions. InitSlicing() initializes all active ele-



ments at the scan line “ $y=from$ ” for incremental slicing. Figure 5 illustrates this process. `RectCopy()` is used to copy a scan line of computed pixel values, sl , into a rectangular buffer called *bitmap* at a location determined by the value of y .

Figure 6 shows the pseudocode for the node process. All variables for the message passing are defined identically with those in the host process but are, of course, stored in different locations. The pseudocode introduces only one new variable, *host*. This variable holds the number of the host process and is assumed to have received its value during the initialization of all processors. (We discussed inner loop functions and variables under “Scan-line volume rendering.”)

Test data

We chose three data sets for the performance tests, where all volume elements are node-valued hexahedra produced by a numerical simulation. For such data, the number of elements chosen for the tests—1,270, 6,250, and 20,975—represents a small, medium, and large data set, respectively. For other representations of volumes, such as voxels, the number of volume elements is usually much higher.

Thus far, we have discussed volume data sets only in terms of the number of elements n . We believe example plots of the test data will also be of interest. For this reason, we have included three color images where blue, yellow, and red regions indicate low, intermediate, and high values, respectively. However, the transparency parameter associated with each scalar interval varies from one picture to another. In all pictures, we tried to facilitate data interpretation by showing how the scalar values relate to fixed visual references, such as terrain or mesh boundaries.

We used two methods for this purpose. The first method, called *reference shading*,^{7,10} is based on a modulation of color saturation. The second method superimposes a volume onto an arbitrary image defined by color and z values, as demonstrated for opaque surfaces in the color images. The latter method¹⁰ is trivial to include in the parallel algorithm and has no noticeable effect on the response times.

Figure 7 shows one geologic layer extracted from a nine-layer reservoir data set. The boundary of the mesh and the geologic faults within the mesh are displayed as grey opaque surfaces. In Figure 8, the reservoir consists of six geologic layers and in-

Figure 5. The task performed by `InitSlicing()`.

Figure 6. The algorithm on each node.

```

/* receive the data set */
Receive(PNUM,ELM,sizeof(struct elm));

for (each new image to be computed) {

    /* receive the parameters for the image */
    Receive(PNUM,TM,sizeof(struct tm));
    Receive(PNUM,DM,sizeof(struct dm));

    /* initialize all elements */
    TransformElements(tm,ne,YMAX,ELM);
    SortElements(y,ne,ELM);

    while (there is more sections) {
        Receive(PNUM,FROM,4);
        Receive(PNUM,TO,4);

        /* process one section */
        InitSlicing(from,ELM);
        for (y=from; y>=to; -y) {
            ActiveElements(y,FIRST,LAST);
            for (j=first; j<=last; ++j) {
                Slice(elm[j],y,I,NI);
                PolyConstruct(i,ni,P,NP);
                for (k=0; k<=np-1; ++k)
                    FillBuffer(p[k],dm,SB);
            }
            for (all points x on the scan line)
                AddContributions(sb,dm,x,SL);
            RectCopy(y,sl,BITMAP);
        }
        /* send the result to the host */
        Send(host,from,4);
        Send(host,to,4);
        Send(host,bitmap,4*(from-to+1)*wx);
    }
}

```

cludes one major geologic fault. Reference shading enhances the mesh boundaries and the exterior part of the fault surface. Figure 9 shows simulated potential temperatures over a Gouraud-shaded terrain model and reference shading applied to the exterior mesh surfaces.

Performance tests

We implemented the innermost loops of Figure 6 as described in detail by Giertsen,⁷ but included the additional procedures introduced in this article for handling large data sets efficiently. We used PVM for message passing.

Our computing environment for the tests consisted of IBM



Figure 7. Oil pressure in a geologic layer, 1,270 volume elements.

RISC System/6000 Model 550 workstations connected to a 16-Mbits/second IBM Token Ring LAN. We used a group of homogeneous workstations for the tests to enable comparison of response times for different numbers of workstations. We repeat, however, that the parallel algorithm has no bottlenecks that would prevent high performance when using a group of heterogeneous workstations.

We performed all tests at night, with no other users logged in and with no major system services taking place. Each test was repeated four to five times to be sure of getting stable results. In the tests, we computed the volumes shown in Figures 7, 8, and 9 at low and high resolution, that is, 512×512 and $1,024 \times 1,024$, respectively. The efficiency analysis indicates the importance of testing the algorithm at different resolutions (see the sidebar). For each image size, we also varied the number of sections ($nprocs \leq nsect \leq 100$).

We ran all tests for two, three, and four workstations—in total, more than 1,000 tests. Figures 10, 11, and 12 show the results. In each figure, the upper and the lower sets of curves correspond to high and low resolution, respectively. We report the results in wall-clock time, since this is the response time of interest to a user. Thus, our timings include processing, communication, and waiting by idle processors.

We also wanted to test parallel performance on loaded workstations. However, defining an experimental setup of scientific value for this purpose is a complex problem. Typical examples of loaded workstations must be defined. All parameters on the workstations and the network affecting the results must be identified and controlled. We could not find proposals on a suitable experimental setup in the computing literature.

Test results

Figures 10, 11, and 12 on the next page indicate that the load balancing is very unstable for small values of $nsect$. The need for dividing the volume into many sections is particularly evident when the test data is drawn at high resolution. Based on all tests, we conclude that values of $nsect$ between 20 and 30 always ensure high parallel performance.

We can draw more general conclusions by deriving a model for the parallel computing time t_{nprocs} , when using $nprocs$ workstations. Equation 1 expresses such a model, where t_{idle} is the maximum time spent by a workstation to wait idly for the others to complete an image, t_1 is the time required to compute an image using one workstation, t_{com} is the time spent for message passing, t_{sys} is the time spent by the operating system to admin-

Figure 8. Pore volume in a complete reservoir, 6,250 volume elements.

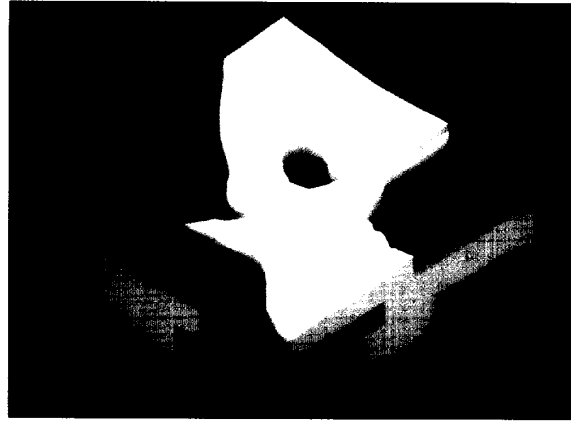


Figure 9. Potential temperature over Barcelona, Spain, 20,975 volume elements.



ister the processing of one section, and t_{ini} is the time required to initialize all elements n_s in a section for incremental slicing.

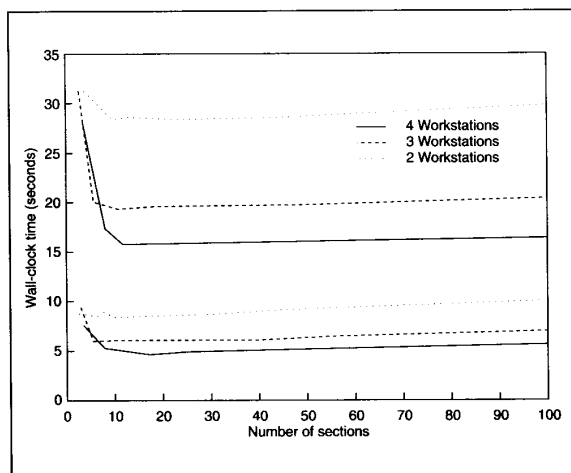
$$t_{nprocs} = t_{idle} + \frac{t_1}{nprocs} + \frac{t_{com}}{nprocs} + \frac{(t_{sys} + t_{ini})nsect}{nprocs}$$

In our case :

$$t_{com} = \left(3t_0 nsect + \frac{16bytes + w_x w_y \cdot 4bytes}{16Mbits/second} \right) \frac{nprocs - 1}{nprocs}$$

$$t_{ini}(n_s) = 9.54 \cdot 10^{-5} n_s + 3.16 \cdot 10^{-5} \text{seconds} \quad (1)$$

In the expression for t_{com} we assume that the transfer of the pointers *from* and *to* is implemented as one single message. Then there will be three messages for each section. First, the host sends the pointers to the node. Then, the node sends the point-



ers back to the host when the section is computed, followed by a message containing the rectangle of pixels. t_0 is the time needed to initialize a message transfer, typically 5 milliseconds.

We introduce the factor $(nprocs - 1)/nprocs$ because the host program and one copy of the node program are usually executed on the same machine. In that case, message-passing between the node and host processes takes virtually zero time; data is simply transferred within the internal storage. We derived the expression for t_{mi} experimentally by running the procedure `InitSlicing()` for many different values of n_s . As expected, the results matched a line equation almost perfectly.

In Equation 1, all operations computed in parallel are divided by the variable $nprocs$. On infrequent occasions, all processors might be ready to communicate results at exactly the same time. In such cases, the term $t_{com}/nprocs$ does not accurately describe reality. Some minor waiting will be introduced.

The goal of parallel scan-line volume rendering is to minimize t_{nprocs} . All curves in Figures 10, 11, and 12 show examples of minimal t_{nprocs} values. We can expect that one such minimum value always exists and that this value mainly depends on the value of $nsect$. For a few sections, the value of t_{idle} is generally high due to poor load balancing. In this case, the term including t_{mi} and t_{sys} is low, since there are few sections to initialize and few interrupts in the processing. For a high number of sections, the opposite is true. We consider modeling t_{idle} and t_{sys} beyond the scope of this article. Equation 1 can still be used to explain the main characteristics of the curves in Figures 10, 11, and 12.

In the stable part of all curves ($nsect > 20$), we can observe a decrease in the slope of the curves as the number of workstations increases. We explain this general trend, which is particularly evident in Figure 12, by the parallel computation of the total overhead introduced by parallel processing. Thus, using more workstations makes the overhead less apparent in the response times.

Another way to plot the test results is first to compute the speedup defined by the ratio between t_1 and observed t_{nprocs} values, and then to plot the speedup as a function of the number of sections. This makes it easier to compare parallel performance for different numbers of workstations, different data sets, and different window sizes. In Figure 13, we have drawn one such plot based on the upper set of curves in Figure 12. We chose to use the temperature data drawn at $1,024 \times 1,024$ resolution, since we are particularly interested in improving the re-

Figure 10. Test results for oil pressure in a geologic layer, 1,270 volume elements. The response times when using one workstation are 52.9 seconds for $1,024 \times 1,024$ resolution and 15.4 seconds for 512×512 resolution.

Figure 11. Test results for pore volume in a complete reservoir, 6,250 elements. The response times when using one workstation are 119.6 seconds for $1,024 \times 1,024$ resolution and 34.4 seconds for 512×512 resolution.

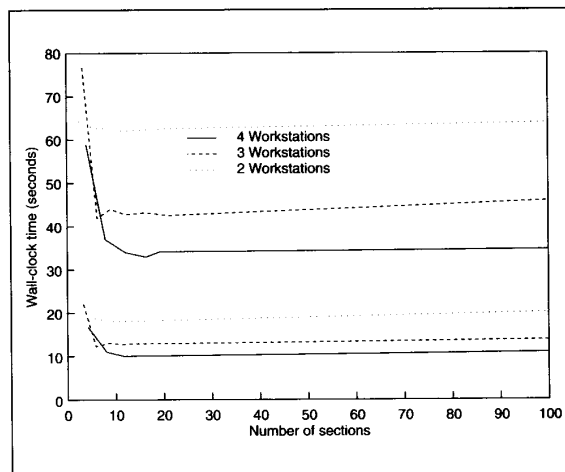
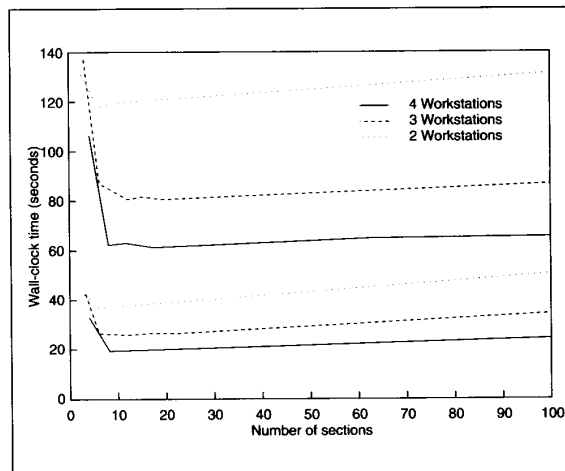


Figure 12. Test results for potential temperature over Barcelona, 20,975 elements. The response times when using one workstation are 232.2 seconds for $1,024 \times 1,024$ resolution and 72.8 seconds for 512×512 resolution.



sponse times for large image size and large data sets. We achieved a speedup as high as 3.83 when using four workstations. This result indicates a potential for further improving performance by using more workstations.

The measurements represented by Figures 10, 11, and 12 can be plotted in many other ways. We leave this as an exercise for the reader. For example, it might be interesting to plot speedup

Figure 13. Speedup results for upper set of curves from the Barcelona temperature image, $1,024 \times 1,024$ resolution.

versus number of workstations for a fixed number of sections or to compare actual timing curves with theoretical timing curves. In fact, we produced the latter curves by setting t_{idle} and t_{sys} in Equation 1 to zero. As expected, the theoretical timing curves and the actual timing curves corresponded well at all points where t_{nprocs} takes the minimum value.

Conclusions

The general framework for parallel scan-line volume rendering presented here can add value to distributed computing environments without demands for additional investments. It achieves high parallel performance by minimizing network traffic and by using a load balancing strategy that ensures all processors finish at almost the same time. The framework is flexible and can easily be extended to include rendering of multilayered semitransparent geometric objects.¹¹

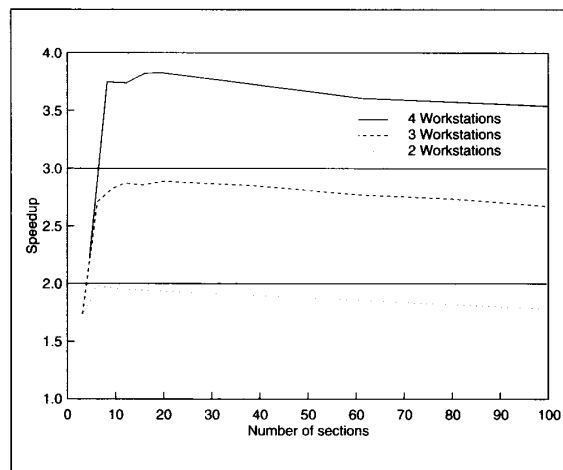
We tested the algorithm under optimal conditions to ensure reproducible test results. In a more realistic situation, the response times will mainly depend on the available CPU capacity, provided that the network is not saturated. As long as there is some CPU capacity available on each workstation, our dynamic load-balancing scheme ensures close-to-optimal utilization of the available resources. □

Acknowledgments

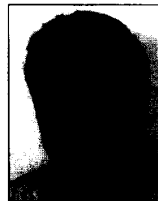
We thank Robert MacKinnon for computing system support and Patrick Gaffney for comments on the manuscript (both at the IBM Bergen Environmental Sciences and Solutions Centre). The Norwegian oil company Statoil provided the reservoir simulation data. Roberto San Jose (University of Valladolid, Spain) and Thomas Flassak (University of Karlsruhe, Germany) provided the air pollution data.

References

1. A. Kaufman, ed., *Volume Visualization*, IEEE Computer Society Press, Los Alamitos, Calif., 1991.
2. A. Kaufman and R. Bakalash, "Memory and Processing Architectures for 3D Voxel-Based Imagery," *IEEE CG&A*, Vol. 8, No. 11, Nov. 1988, pp. 10-23.
3. R. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics*, Vol. 22, No. 4, 1988, pp. 64-75.
4. J. Petersen, "Introduction to Programming on Distributed Memory Multiprocessors," *Computer Physics Communications*, Vol. 73, No. 1-3, Dec. 1992, pp. 72-92.
5. M. Potmesil and E.M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 69-78.
6. A. Beguelin et al., "A Users' Guide to PVM Parallel Virtual Machine," Tech. Report TM-11826, Oak Ridge National Laboratory, Oak Ridge, Tenn., 1991.
7. C. Giertsen, "Volume Visualization of Sparse Irregular Meshes," *IEEE CG&A*, Vol. 12, No. 2, Mar. 1992, pp. 40-48.

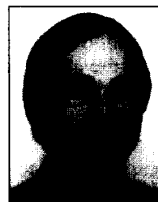


8. M. Garrity, "Raytracing Irregular Volume Data," *Computer Graphics*, Vol. 24, No. 5, Nov. 1990, pp. 35-40.
9. P. Shirley and A. Tuchman, "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics*, Vol. 24, No. 5, Nov. 1990, pp. 63-70.
10. C. Giertsen, "Creative Parameter Selection for Volume Visualization," *J. Visualization and Computer Animation*, Vol. 3, No. 1, Jan.-Mar. 1992, pp. 1-11.
11. C. Giertsen and A. Tuchman, "Fast Volume Rendering With Embedded Geometric Primitives," in *Visual Computing—Integrating Computer Graphics with Computer Vision*, T.L. Kunii, ed., Springer Verlag, Tokyo, 1992, pp. 253-271.



Christopher Giertsen recently joined Christian Michelsen Research in Bergen, Norway, as a senior scientist. For the past six years, however, he worked in scientific visualization research at the IBM Bergen Environmental Sciences and Solutions Centre. The work presented in this article was performed at IBM. His research interests include 3D reconstruction from serial sections, volume rendering, user-interface aspects of visualization, and color models for computer graphics.

Giertsen received his Dr. Philos. degree in computer science from the University of Bergen. His e-mail address is chrisgie@cmr.no.



Johnny Petersen is a senior scientist at the IBM Bergen Environmental Sciences and Solutions Centre in Norway. His research interests include numerical modeling on parallel computers and parallel scientific visualization.

Petersen received his BS and PhD in physics at Brigham Young University in 1976 and 1982, respectively. His e-mail address is johnny@bsc.no.

Readers can contact Giertsen at Christian Michelsen Research, Fantoftveien 38, 5036 Fantoft, Norway, and Petersen at the IBM Bergen Environmental Sciences and Solutions Centre, Thormøhlensgate 55, 5008 Bergen, Norway.