INTELLIGENT PARALLEL VOLUME RENDERING USING VIEW COHERENCE ON MIMD ARCHITECTURES

Selim Temizer, Ömer Sinan Saraç, Veysi İşler

Department of Computer Engineering
Middle East Technical University
Ankara Turkey
[e106193|e106183|isler] @ceng.metu.edu.tr

Abstract

Although volume rendering is a powerful computer graphics technique for visualizing data represented at three spatial dimensions, it is computationally intensive. Parallel Computing represents the most promising solutions for computationally expensive problems. This paper aims to introduce some new concepts and proposes a new method for parallel volume rendering on MIMD architectures. The method described uses dynamic load balancing techniques to achieve higher efficiency and especially focuses on efficient progressive redistribution of the data during the consecutive rendering of animation frames by taking view coherence into consideration. The method is named as *intelligent* because most of the necessary actions for rendering are automatically handled by decisions locally made on each processor. Almost all global decisions such as data redistribution scheme and synchronization are replaced with local decisions. The method also incorporates the overlapping of the interprocessor messaging times with useful computation times.

Keywords : Computer Graphics, Parallel Volume Rendering, Dynamic Load Balancing, On-Demand Redistribution, View Coherence

1. Introduction

One major application of visualization in scientific computing is the visualization of scalar functions of three spatial variables. This kind of volume data is confronted very often in scientific and engineering studies. A voxel volume is either produced by a mathematical function, such as in computational fluid dynamics, or they are collected from the real world, as in medical imaging. Sources of our data used in the test of the algorithm are from medical imaging. We have worked on both CT and MRI data. The data sets used in the test phase of this study can be obtained from Internet (ftp: wustl.carlos.edu/~dist/imagelib).

Volume rendering has been shown to be a useful technique for visualizing the set of data defined in three spatial dimensions. The basic idea is that a viewer should be able to perceive the volume from a rendered projection on the view plane. We can view the whole volume or a subset of it by assigning different opacity and color to the different parts of the volume. This process is called *selective rendering*.

The rendering method selected in this work is Ray Casting. It is based on shooting a ray from each pixel of the viewing window and trying to find the final color of that pixel by accumulating the contributions along the ray path. Since the calculation of a pixel color is independent of any other pixel calculations, there is a natural parallelism in this process. Instead of using only one processor, we may easily utilize many processors for calculation of the pixel colors of different regions of the viewing window and combining the results. The only restriction in this case is that; the processor, which is in charge of finding a pixel color, should have all the voxels (worst case situation – assuming high transparency in the data) along the ray path. And if data is distributed among the processors, some interprocessor data transfers should be done. This is called redistribution, which is the most important bottleneck when object space decomposition is used in any parallel rendering algorithm.

This work proposes a method for the problem described above, namely for parallel volume rendering using view coherence on MIMD architectures. The method can also be generalized for many other parallel application domains whose data and functional dependencies resemble the situation in volume rendering using ray casting method. Some parts of the method can easily be applied to most parallel applications regardless of whether the dependencies resemble the mentioned situation above or not.

In this work, the Master - Slave Paradigm is used as the parallel approach to the problem. The method is named as *intelligent* because of automatic handling of many necessary actions for volume rendering that is achieved by localized effective decision making capabilities supplied to both the master and the slave programs. Also when parallel processing is taken into account we confront with the problems; load balancing, efficiency and scalability. Load balancing may be implemented statically or dynamically. The algorithm in this paper tries to exploit the profits of both methods.

This algorithm also utilizes image space decomposition. Redistribution of the data is achieved in parallel with rendering process. Required voxels are obtained as needed by sending requests to the neighbor processors. Therefore the algorithm offers progressive redistribution of data during consecutive frame renderings. This will be referred to as *On-Demand Redistribution* in this paper. Instead of waiting for an answer to the data request, the algorithm continues with another ray from an array of rays under processing, aiming to overlap the message waiting and computation times.

This paper is organized as follows: the next section contains related work on parallel rendering. In Section 3, the behavior of the master is presented. Section 4 and Section 5 define important concepts and variables that are referred to in Section 6, which describes the behavior of the slave. Section 7 presents the results. Finally, Section 8 derives the conclusion. Appendix A and Appendix B includes the main algorithm of the slave and the most important function of the slave, respectively.

2. Related Work

Static load balancing method for parallel ray tracing is studied by Thierry Priol and Kadi Bouatouch [1]. Todd Elvins conducted a survey of algorithm for volume visualization [2]. Fuchs proposed a technique for distributing the z-buffer hidden surface removal algorithm over a distributed-memory MIMD architecture [3]. Cleary, et al. give an algorithm for ray tracing which utilizes a world-space decomposition on a distributed-memory MIMD system [4]. The rays are represented as messages passed between the processors. An adaptive world-space decomposition technique is studied by Dippe and Swensen to achieve better load balancing [5]. A parallel ray-tracing algorithm on a distributed memory MIMD architecture using image-space decomposition is presented by Badouel [6]. Whitman analyses several image-space decomposition methods and scheduling strategies for the scan conversing of polygons on a shared-memory MIMD machine [7]. A detailed analysis of the overhead incurred in the parallelization is presented. Levoy presented a paper on Efficient Ray Tracing of Volume Data [8].

3. Master Behavior

The master program is responsible for the following actions:

1. Gathering the necessary graphical information from the user such as the size and position of the Viewing Window, the color and position of the light source, etc.
2. Informing the slaves about the gathered information.
3. Distributing the work of computation of the color of the pixels constituting the final picture (frame) to the slaves in a *clever* fashion.
4. Gathering the results from the slaves and displaying the frame.

The most important action performed by the master from the point of view of load balancing is the 3rd one. The master assigns non-equal regions to the slave programs. The decision about the amount that will be assigned to each slave is determined in the following way: After each frame computation finishes, the slaves not only send the final colors of the pixels that they computed, but also the computation weights of each pixel to the master. When the next frame will be drawn, the master takes the weight of the pixels into account and tries to assign nearly equally weighted regions to each slave. An important assumption that should be made in order to utilize view coherence is that, the movements of the viewing window are always in small amounts. Therefore, the previous pixel weights will be a very good approximation for the next frame pixel weights. It should be noted that for the very first frame, the master has no idea about the weights of the pixels and assigns equal regions to the slaves. The clever decisions start with the second frame and continue afterwards.

4. Some Important Concepts

- ***Database*** : Each slave has its own database that contains the voxel values that the slave has at any moment.
- ***Voxels*** : Each voxel has a value and a boolean attribute called "Movable". If this attribute is TRUE, then, when a VOXEL_REQUEST_Message is received by this slave with Voxel_Request_Mode = MOVE, the slave not only sends the voxel to the requesting slave, but also removes the voxel from its database. If it is FALSE, the voxel is kept in the database and a copy of the voxel value is sent to any requesting slave even if the Voxel_Request_Mode of any VOXEL_REQUEST_Message is MOVE. The messages will be discussed later.
- ***Process*** : The term *process* is used to indicate the processing of any one of the pixels that constitute the final picture. It should not be confused with the process concept in operating systems, etc. The processing of one pixel is simply the determination of the final color of that pixel by using ray casting method on the volume data in our case. Each slave has a fixed number of processes that is determined when the slave is created. The slave resumes the execution of its processes one by one, runs the process it resumed for some time, saves the process status, preempts the process and resumes the next process in turn. Having more than one process provides the following: If at some time during its execution, any process needs some voxels that the slave does not have in its database, the slave requests these values from other slaves. Instead of waiting for the values for that process, the slave continues with computation of some other pixels in its region, that is; the slave stops running this process and resumes the other processes one by one. If the number of processes is large enough, then, when the turn comes to the requesting process, the values may have come during the computation period of the other processes. By this way the message waiting time and computation time are

overlapped. The number of processes should not also be too large. This degrades the efficiency of dynamic load balancing. The dynamic load balancing is provided by slave-slave cooperation and explained later.

Each process has some attributes. In the pseudocodes given in appendices, any process is indicated as Process[i] meaning that this is the ith process in the array of processes. Any attribute of ith process is indicated as Process[i].<Attribute_Name> using the dot notation. The important process attributes and their brief explanations are given below:

- ***Status*** : The status may have two values which are RUNNING, indicating that the processing of the pixel assigned to this process is not finished yet, and FINISHED, indicating that this process does not have a pixel processing assigned at that moment.
- ***Execution_Step***: There are four steps in the execution cycle of a process. These are INITIALIZE_STEP, DATA_REQUEST_STEP, DATA_WAIT_STEP and COMPUTE_STEP. The tasks that are performed at each step are explained in the pseudocode at the appendices in details.

5. Important Variables

- ***Number_Of_Processes*** : Keeps number of processes of that slave.
- ***Number_Of_Running_Processes*** : Keeps number of processes with status RUNNING, at any time.
- ***Pixels_Remaining*** : This variable keeps the number of pixels still remaining to be computed for that slave. The pixels that are assigned to processes and are now currently being processed are not counted in this variable.
- ***Minimum_Cooperate_Limit*** : If the Pixels_Remaining is below this variable, then when a neighbor offers help, instead of sharing the work, the slave refuses the help offer. If the message sending time needed for sharing your work is higher than the time needed to compute the pixels, then do not share your work with your neighbor.
- ***Waiting_For_Help_Reply*** : If TRUE, then the slave does not send any other help offers to its neighbors but waits for the result of the last send offer.
- ***Computation_Result_Sent*** : If TRUE, then it means that the slave finished its own work and sent the result to the master, and now it can offer help to its neighbors.

6. Slave Behavior

In the following discussions, the slaves that are assigned adjacent regions are referred to as neighbor slaves. The slave whose number is one less than the number of a slave is the up neighbor of that slave. The other neighbor is the down neighbor. Each slave mostly communicates with its neighbors. Before proceeding further, the reader is expected to refer to Appendix A and Appendix B for an overall picture.

Some properties of the slave behaviors are as follows: A slave first tries to finish the pixels in its region. If it finishes, it sends the result to the master and offers help to its up and down neighbors in turn. If any one of them accept the offer, the accepting slave also sends half of its work remaining (Half of Pixels_Remaining) to the offering slave. The slave has now again a work to do and the cycle restarts.

The slaves help only to their up and down neighbors. They do not offer help to other slaves. This is due to the reason that neighbor slaves have some voxels in common and data transfer may not be necessary if work is shared with a neighbor rather than a remote slave. But if the work shared is hard, that is; if pixels shared are more than

Minimum_Cooperate_Limit, then the help offering slave will also accept the help offer coming from its other neighbor, and the hard work will be shared in a rippled fashion. This provides full dynamic load balancing on the fly.

The slaves offer help only once. If they are refused, they do not offer help again to the refusing neighbor. If the refusing neighbor is assigned some work, it should explicitly notify its neighbors that it has now some work and will be pleased if help is offered. By this way, all slaves are automatically synchronized after a frame computation is finished, and no global synchronization is needed.

The data redistribution among the slaves for each frame is also automatically handled. When a slave needs some voxels during the color computation for a pixel and does not have them in its own database, it requests the values from one of its neighbors. This is the On-Demand Redistribution Scheme used in this algorithm that is mentioned in the previous sections. The request is sent to the most promising neighbor. That is; if the voxel is needed for a pixel which is in the upper half of the region that this slave should calculate, then the voxel is requested from the up neighbor. Otherwise, it is requested from the down neighbor. This is done with the utilization of view coherence in mind. The master-slave and slave-slave communications are done by messages. The important messages are explained below:

- ***NEW_COMPUTATION_Message*** : The master sends this message to the slaves to inform them about the graphical settings and to assign them their regions that they should compute.
- ***VOXEL_REQUEST_Message*** : If a slave does not have a voxel in its database and needs it for the computation, it sends a request message to the most promising neighbor. This is explained above in details. When a slave receives a request message from one of its neighbors, it first checks the voxels requested, and sends the voxels to that slave. For the values that it also does not have in its database, it creates another request message and sends it to its other neighbor. The message also contains the mode of the voxel request. If the request mode is MOVE, and the slave receiving the message also believes that it will not need the requested voxels for this frame calculation, it both sends the value and removes the value from its database. In all other cases, a copy of the value is sent, and value remains in the database.
- ***HELP_OFFER_Message*** : The slave finishing its own region sends this message to its neighbors for sharing their work if they accept. The slave receiving this message refuses offer if it does not have any work or if the Pixels_Remaining is below Minimum_Cooperate_Limit.
- ***NOTIFY_WORK_Message*** : If a slave knows that its neighbors will not offer help to it anymore, and it is somehow assigned a work and Pixels_Remaining is greater than Minimum_Cooperate_Limit, then it notifies its neighbors about this situation by this message. The slave receiving this message may offer help or not depending on its own work situation.

The pseudocodes for the most important parts of the slave program are given in appendices. The comments start with /* sequence and end with */ sequence. Some commands are borrowed from C Programming Language such as Return, Switch, etc.

7. Results

In order to demonstrate the properties of the algorithm and to present the results obtained, two runs will be explained now. In the first run, 35 consecutive frames are drawn. Before each frame is drawn, the viewing window is rotated 5 degrees around the object rendered. The amount of time needed to render these 35 frames is shown in Figure 1. In this run, we see that after the first frame is drawn, the master collects enough information about the object rendered, and effectively plays role in dynamic load balancing for the consecutive frames.

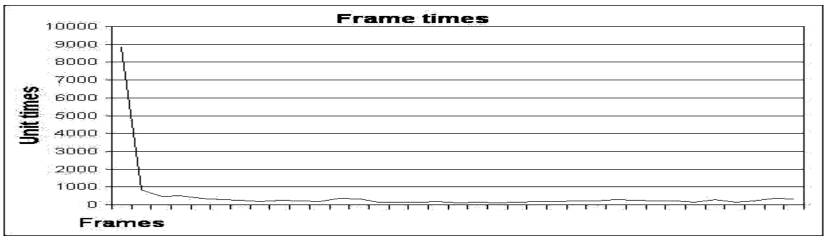

Figure 1. Frame Rendering Times

In the second run, 3 slaves are used. The viewing window is 200x200 pixels large. The Number_Of_Processes for all slaves are 40. And the Minimum_Cooperate_Limit is set to 10. The scenario of the second run is as follows, first, a viewing position is selected and the first frame is drawn. The image obtained is shown in Figure 2. Since the master has no idea about the weights of the pixels for the first frame, it assigned equal portions of the viewing window to each slave. Figure 3 shows which pixels are actually computed by which slave as a result of cooperation. Each different color indicates one of the slaves. In Figure 4, we see the weights of the pixels for the first frame. The increasing computational complexity is indicated with the increasing darkness of the points.

After the first frame, the viewing window is rotated 5 degrees and the second frame is drawn. This time the master took weights into account, and distributed the regions accordingly. Figure 5 shows which pixels are actually drawn by which slave in the second frame. And for the third frame, the viewing window is kept fixed. This time, since the weights of the pixels are not guessed closely, but known exactly, the region assignment of the master was very successful and only two cooperation existed among the slaves. Figure 6 shows which pixels are actually drawn by which slave in the third frame. And lastly, to see the results of cooperation for more than 3 slaves, you can refer to Figure 7 which shows the result of cooperation in a 10 slave run for the first frame.

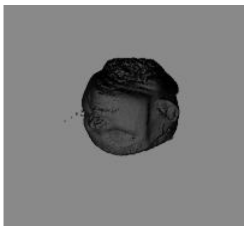

Figure 2

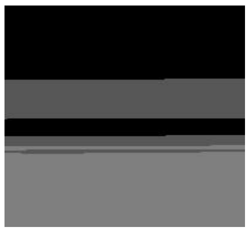

Figure 3

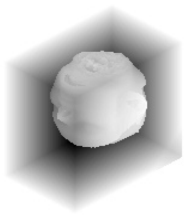

Figure 4

Table 1 summarizes the second run by indicating the number of pixels assigned to each slave by the master and the actual number of pixels computed by each slave.

	Frame 1		Frame 2		Frame 3	
Slave	Assigned	Computed	Assigned	Computed	Assigned	Computed
1	13333	16465	16474	16267	16174	16174
2	13333	9503	9656	11484	10169	10135
3	13334	14032	13870	12249	13657	13691

Table 1

Figure 5

Figure 6

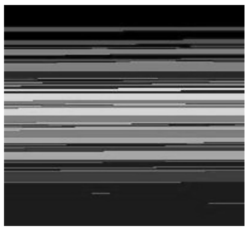
Figure 7

8. Conclusion

This work focuses on parallel volume rendering that is incorporated in an animation system. Parallel processing issues, in this case, get more complicated. The object space data is to be distributed among the processors in such a way that redistribution of the data during frame rendering is minimized and load balance is maintained. This study has resulted in some methods to overcome the mentioned problems. These methods utilize the terms such as *Intelligent Rendering*, *On-Demand Redistribution Scheme* and *Animation Frame Rendering*, which are aimed to be introduced as the minor goal of this research. Another important outcome of this study is to propose a way to overlap the redistribution time with the computation time at the processors.

References

[1] K. Bouatouch and T. Priol. “Static load balancing for a parallel ray tracing on a MIMD hypercube”. In The Visual Computer (5): 1989, 109-119.

[2] T. Elvins. “A Survey of Algorithms for Volume Visualization.” In Computer Graphics, volume 26, number 3, August 1992, 194-200.

[3] H. Fuchs. “Distributing a visible surface algorithm over multiple processors.” In Proceedings of ACM, October 1977, 449-451.

[4] G. Cleary, B. Wyvill, G. M. Birtwistle, and R. Vatti. “Multiprocessor ray tracing.” Technical Report 83/128/17, University of Calgary, 1983.

[5] M. Dippe and J. Swensen. “An adaptive subdivision algorithm and parallel architecture for realistic image synthesis.” In Computer Graphics, pages 149-158. ACM Siggraph ’84 Conference Proceedings, 1984.

[6] D. Badouel and T. Priol. “An efficient parallel ray tracing scheme for highly parallel architectures.” In Proceedings of the Fifth Eurographics Workshop on Graphics Hardware, September 1990.

[7] S. Whitman. “Multiprocessor Methods for Computer Graphics Rendering.” Jones and Bartlett, October 1991.

[8] M. Levoy. “Efficient ray tracing of volume data.” In ACM Transactions on Graphics, Volume 9, No. 3. July 1990, 245-261.

Appendix A. The slave behavior may be summarized by the pseudocode below:

```
Program : Slave_Program
{  While ( TRUE )     /* Main Slave Loop ; Slave is always somewhere in this loop */
   {  While ( There are messages arrived and waiting to be handled )
      {  Handle message in turn ;  }
      if (  Number_Of_Running_Processes  >  0  )
      {  for  i = 1  to  Number_Of_Processes     {  Resume_Process( i ) ;  }  }
      else   {   if ( Computation_Result_Sent = FALSE )
```

```
                {  Send Computation Result (colors and weights of the pixels) to Master ;
                   Computation_Result_Sent = TRUE ;    }
                if (  Waiting_For_Help_Reply = FALSE )
                { Send help offer to up and down neighbors in turn, if necessary } } } }
```

Appendix B. The Resume_Process function is the most important function providing the overlapping of message waiting time and computation time.

```
Function : Resume_Process ( Integer : Process_No )
{  if  (  Process[Process_No].Status = FINISHED  )   {  Return ;  }
   switch  (  Process[Process_No].Execution_Step  )
   {  case  INITIALIZE_STEP :
         Determine starting point of ray originating from this pixel ;
         Find "Ray - Data_Box intersection" points ( Entry point and Exit point ) ;
         if  (  No intersection  )
         {  Color of this pixel = Background color ;
            Weight of this pixel = Weight of "Ray - Data_Box intersection" calculation ;
            Number_Of_Running_Processes = Number_Of_Running_Processes - 1 ;
            if  ( Pixels_Remaining  >  0  )
            {  Initialize this process with the pixel in turn waiting for being computed; }
            else   {  Process[Process_No] = FINISHED ;  }
            Return ; } /* if there is intersection, execution continues downwards */
      case  DATA_REQUEST_STEP :
         if  (  I have some or all of the voxels for calculations at Current_Position )
         {  Set Movable attribute of those voxel values to FALSE ;  }
         if  (  I don't have some or all of the voxels for calculations at Current_Position )
          {  Create a waiting list of the voxel values that I do not have and are not
                marked as REQUESTED ;
            Mark those voxel values as REQUESTED ;
            Send VOXEL_REQUEST_Message for those voxels ;    }
         if  (  All needed voxels were ready and no request message is sent above  )
         {  Process[Process_No].Execution_Step = COMPUTE_STEP ;  }
         else   {  Process[Process_No].Execution_Step = DATA_WAIT_STEP ;  }
         Return ;
      case  DATA_WAIT_STEP :
         Check each voxel in the waiting list, remove those whose values are received ;
         if  (  Waiting list is not empty after the check  )      {  Return ;  }
      case  COMPUTE_STEP :
         Update Weight for this pixel ; /* add weight of computations done in this step*/
          Perform the graphical calculations at this point for determining the local color
            contribution, opacity, etc.
         Advance ray along the ray path ;
         if  (  This pixel computation is finished  )
         {  /* Weight and color for this pixel is known now */
            Number_Of_Running_Processes = Number_Of_Running_Processes - 1 ;
            if  ( Pixels_Remaining  >  0  )
            { Initialize this process with the pixel in turn waiting for being computed ; }
            else   {  Process[Process_No].Status = FINISHED ;  }     }
         else   {  Process[Process_No].Execution_Step = DATA_REQUEST_STEP ;  }
         Return ; }   }
```