1994

Scalar Field Modeling & Visualization on the Intel Delta

Chandrajit Bajaj

Kwun-Nan Lin

Report Number:
94-039

Scalar Field Modeling & Visualization on the Intel Delta

Chandrajit Bajaj and Kwun-Nan Lin
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-94-039
June, 1994

Proc. of the 1994 INTEL Supercomputer Users Group, San Diego, July 1994.

Scalar Field Modeling & Visualization on the Intel Delta*

Chandrajit Bajaj
Department of Computer Science,
Purdue University,
West Lafayette, IN 47907
Email: bajaj@cs.purdue.edu
and

Kwun-Nan Lin
Department of Electrical Engineering,
Purdue University,
West Lafayette, IN 47907
Email: kwunnan@ecn.purdue.edu

Abstract

We describe the parallelization of 3D scalar field modeling/visualization programs on a MIMD machine using message passing. The 3D scalar field is a 3D cuberille volume of scalar intensities such as parallel image slices of CT (Computed Tomography) or MRI (Magnetic Resonance Imaging) data. The cuberille volume is divided into small subcubes. Each subcube is approximated by a trivariate cubic polynomial function. These polynomial functions are constructed to locally achieve C^1 continuity on the shared faces of the cubes, and provide a compact interior representation (model) of the entire volume image data. The parallelization is based on a "multi-host" partition scheme. Several possible partition methods are discussed, and our method is justified in terms of efficiency and low I/O overhead. Several strategies were implemented to eliminate or alleviate the problems usually associated with "host-slave" partition scheme. Our implementation shows efficient utilization of all nodes with good scalability.

1 Introduction

Measurement-based volumetric data sets arise for example from medical imaging (Computed Tomography – CT, Magnetic Resonance Imaging – MRI), atmospheric, geological, geophysical measurements. Synthetic volume data sets are generated for example by computer based simulation such as meteorological, thermodynamic simulations, finite element stress analyses and computational fluid dynamics. Modeling the information contained in these, typically huge, data sets via trivariate polynomial finite element approximations provides mechanisms to allow querying, interaction and manipulation. Volume visualization provides mechanisms to express information contained in these, typically huge, data sets via translucent displays of isosurfaces or volume renderings – the challenge, of course, lies in making these images easy to understand.

Previous related work on MIMD architectures has concentrated on the parallel rasterization of polygons, see for e.g. [6, 3, 4]. In this paper we are involved with the parallel generation of polygons representing different isocontours (isosurfaces) of volume images (3D reconstruction). Our reconstruction approach differs from others (for .e.g [5]) in that we use higher order approximation schemes to yield both a compact C^1 smooth reconstruction and a concise analytic representation (model) for post volume modeling manipulations, volume deformations, etc. Generating polygon isosurfaces from such C^1 reconstruction also gets rid of polygon shading artifacts such as Mach ringing.

We utilize standard graphics hardware for interactive manipulation and viewing of the isosurfaces. Rasterization of the polygons using the parallel machine would be a desirable feature for interactive visualiza-

*Supported in part by NSF grants CCR 92-22467, DMS 91-01424, AFOSR grants F49620-93-10138, F49620-94-1-0080, ONR grant N00014-94-1-0370 and NASA grant NAG-93-1-1473.

tion once a high speed connection is available at Purdue between the MIMD machine and a graphics front-end.

In §2 of this paper we present details of trivariate cubic polynomial approximation computations for dense cuberille data (CT/MRI). In §3 we briefly describe the current Intel computing resources at Purdue. In §4 we discuss the pros and cons of different data and computation partitioning schemes. In §5 elaborate on the "multi-host" partition scheme that we used and provide implementation details. Finally, in §6 we present several tables of timing results from our implementation and experiments.

2 Modeling via Trivariate Cubic Approximation

The input is a set of CT slices of certain XY resolution (such as 512 by 768). Each pixel is associated with a gray level. The number of slices, which form the Z axis, might range from a hundred to several hundreds. We construct piecewise C^1 approximation functions $F(x,y,z)$ such that the function value of every pixel is equal or close to its gray level. The desired output for interactive visualization are sets of polygons. Each set corresponds to an isosurface $F(x,y,z) = c$ corresponding to different anatomical regions. See for example Figures 1,3.

Bernstein-Bezier (BB) forms are used to represent the cubic trivariate polynomial approximation functions. $F(x,y,z) = \sum_{i=0}^{m}\sum_{j=0}^{n}\sum_{k=0}^{q} w_{i,j,k} B_i^m(x) B_j^n(y) B_k^q(z)$; here $B_i^n(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}$ The polynomial function is trivariate cubic if $(m,n,q) = (3,3,3)$. The C^3 data of a pixel is defined as:

$f, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}, \frac{\partial^2 f}{\partial x \partial y}, \frac{\partial^2 f}{\partial y \partial z}, \frac{\partial^2 f}{\partial x \partial z}, \frac{\partial^3 f}{\partial x \partial y \partial z}$ and is computed using finite difference formulas. Let Δx, Δy and Δz be the unit distances of the grid in X, Y and Z directions. Here Z axis is formed by slices. Figure 2 shows an example to compute C^3 data from 2 slices S_0 and S_1. The C^3 data of vertex (1,1,0) is obtained as:

$\frac{\partial f}{\partial x} = \frac{f(2,1,0) - f(0,1,0)}{2\,\Delta x}$, $\frac{\partial f}{\partial y} = \frac{f(1,2,0) - f(1,0,0)}{2\,\Delta y}$,

$\frac{\partial f}{\partial z} = \frac{f(1,1,1) - f(1,1,0)}{\Delta y}$

$\frac{\partial^2 f}{\partial x \partial y} = \frac{f(2,2,0) - f(0,0,1)}{2\,\Delta x\, 2\Delta y}$, $\frac{\partial^2 f}{\partial y \partial z} = \frac{f(1,2,1) - f(1,0,0)}{2\,\Delta y\, \Delta z}$,

$\frac{\partial^2 f}{\partial x \partial z} = \frac{f(2,1,1) - f(0,1,0)}{2\,\Delta x\, \Delta z}$

$\frac{\partial^3 f}{\partial x \partial y \partial z} = \frac{f(2,2,1) - f(0,0,0)}{2\,\Delta x\, 2\Delta y\, \Delta z}$

Paper [2] gives mathematical proofs of the C^1 piecewise continuity of both a (2,2,2) and (3,3,3) trivariate

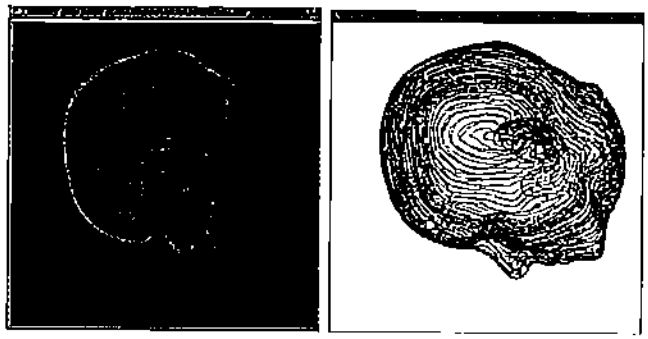

Figure 1: A Stack of C^1 Smooth Cubic Spline Contours which are Iso-Curves from a Stack of MRI Data Slices

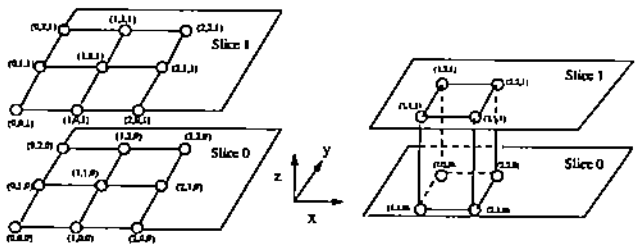

Figure 2: Two Slices of a Cuberille Volume, and a Subcube between these Slices

polynomial approximation. These piecewise higher order approximations are generalizations of the C^0 continuous piecewise linear or trilinear approximations [5, 7]. Here we concentrate on the parallelization of these computations and the generation of isosurfaces.

The algorithm consists of the following steps:

Step 1: *read 2 slices S_0 and S_1 from disk*

Step 2: *compute C^3 data of all pixels located at the corner of the user defined grid from two slices S_0 and S_1.*

It is very computation expensive if the grids cover all pixels. A large grid requires much less computation, but it loses detailed image information since most pixels are not used. The largest grid with every pixel used is 4×4 pixels. Only one quarter of all pixels are at the corner of 4 x 4 grids. The rest pixels are still used in the computation of C^3 data.

Step 3: *read another slice S_2*

Step 4: *similar to step 2, compute C^3 data from three slices S_0, S_1 and S_2*

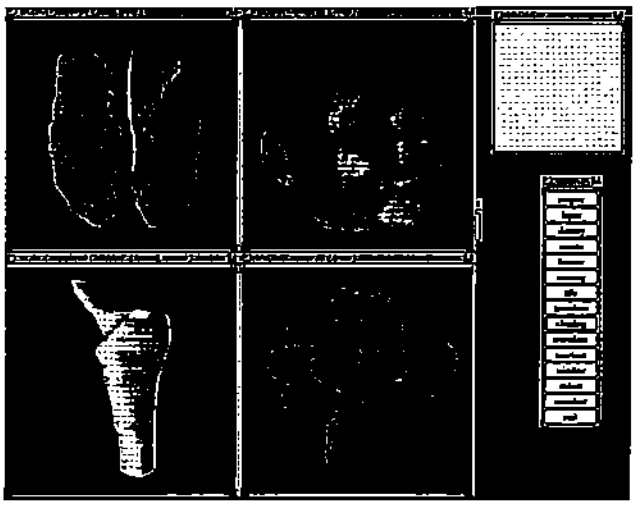

Figure 3: Shaded Display of a Head, Brain, Femur and Lungs which are Isosurfaces of Volume Cuberille MRI Data

The calculation of C^3 data from three slices is similar to that of two slices. It uses the lower slice instead of itself while ∂z is involved. Of course all Δz is replaced by $2\,\Delta z$.

Step 5: *fit a trivariate cubic (3,3,3) function $F(x, y, z)$ to each subcube*
Now we have two slices with C^3 data to form subcubes (Figure 4). Every grid of a slice with its matching grid on the other slice form a subcube. Each subcube consists of 8 vertices, and each vertex has 8 pieces of C^3 data. So there are 64 equations to solve 64 parameters of $(3, 3, 3)$ interpolation in the least square sense.

Step 6: *for each node, compute the polygons of isosurface corresponding to certain constant c such that $F(x, y, z) = c$*

Step 7: *discard one slice and go to step 3 to read a new slice*

3 Computing resources

The machine used in this project was Intel's iPSC/860 installed at Purdue University with 16 nodes and binary compatible with the Touchstone Delta at Caltech. The Paragon that Purdue University is installing will have 128 nodes and be program compatible with the iPSC/860 and Delta machine.

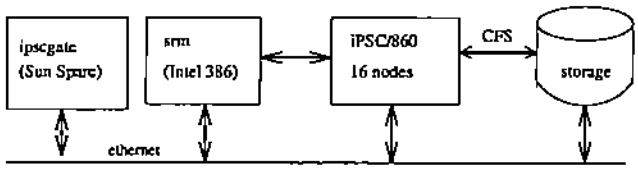

Figure 4: The iPSC/860 system configuration at Purdue

The following description is restricted to our experiments with the iPSC/860 located at Purdue University. Our connection time to the Delta was limited to sporadic use and then only for short durations.

Each node of iPSC/860 is an independent processor with its own local memory (16 M bytes). There is no shared memory. Information exchange is achieved by message passing through a high speed interconnection network. Each node has access to Internet by TCP/IP. All 16 nodes shares a same IP address.

The programs run by each nodes are loaded by a system resource manager (*srm*) which is an Intel 80386 class machine. The *srm* is mainly used to allocate nodes, load programs, kill nodes' processes and deallocate nodes. The program development is performed in a faster machine (Sun Sparc) named *ipscgate*. *srm* and *ipscgate* share the same file systems by NFS. Each node can access the storage by iPSC concurrent file system (CFS). The iPSC user guide states that the CFS allows *several* (perhaps not all, I guess) nodes to access the storage simultaneously. See also Figure 4.

4 Partitioning Schemes

The algorithm in §2 is for a single processor machine. Different partition methods can be utilized to divide the work load. There are at least three classes of partition methods: symmetry, host-slave and multi-host models.

4.1 Symmetry Partitions

In symmetry partitioning each node (processor unit, PE) performs the same job. The volume of CT slices is divided into sub-volumes, and each node processes a sub-volume independently. This static partition usually has problems of unbalanced load and high I/O overhead. The processing time of each sub-volume might be very different, and it is difficult to utilize idle nodes to help busy nodes using a symmetry partition. The cause of high I/O overhead is that a

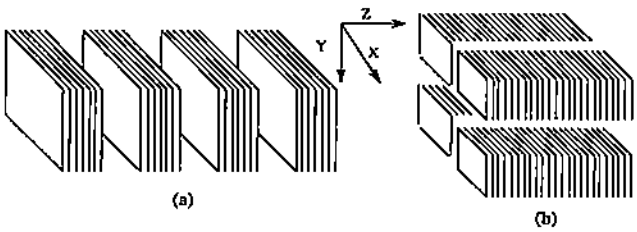

Figure 5: Symmetry Partitions along (a) Z axis, and (b) X and Y axis

slice might be read more than once by different nodes. This partition suits the situation where the slice number is large, and the processing time of each slice does not vary much.

One always incurs higher disk read overhead if the computation requires shared slices. If the partition involves the Z direction such as in Fig. 4.a, some nodes obviously are not used when the node number is more than the slice number. Since four slices (or three slices at the top or the bottom of the volume) are required for the calculation of C^3 data of two slices, the boundary slices between sub-volumes need to be read more than once. For example, if each sub-volume contains only one slice, every slice except the top and bottom slices will be read four times by different nodes.

If the partition involves X, Y or both directions such as in Fig. 4.b, it still has high I/O overhead. Each sub-volume contains a certain region of every slice. The time to read a small region from a whole slice is not proportional to its size. The data proceeding the desired region need to be skipped or scanned. If the slice is in compressed form, the data proceeding the desired region need to be read and uncompressed. So, it takes the same time to read only the last pixel as to read the whole slice of compressed form.

The I/O overhead could decrease efficiency of each node if the concurrent file system (CFS) supports very limited number of simultaneous disk access. Some nodes need to wait for I/O access if they try to access I/O at the same time.

4.2 Host-slave and Multi-host Partitions

Host-slave partition has one host to assign jobs to other nodes. The load of the slaves are usually balanced because the work load of each slave is obtained dynamically. But this partition incurs communication overhead, bottleneck, idle host and idle slaves.

It requires communication overhead because data is transferred from the host to slaves upon request. The slave nodes require the host to always listen to them to achieve fast response. If the host is doing something, such as reading files, other than listening to slave, the slave nodes become idle since no data available from the host. The host causes a bottleneck if it has insufficient computing power compared to its ability to hand out data. On the other hand, it idles if there is no request from slaves during certain time periods.

The multi-host partition entails multiple hosts to reduce the bottleneck incurred from insufficient host computing power. All nodes are divided into groups. Each group has one host and several slaves, and it process a subset of slices. Basically the multi-host partition is a hybrid of host-slave and symmetry partitions. If there is only one group, it is a pure host-slave model. It becomes a symmetry partition if each group has only one node which is doing all the work. However, the multi-host model can re-assign the nodes of a finished group as slave nodes to other groups. Thus the load of each node is better balanced compared to that of the symmetry partition.

5 Implementation Details

Besides, the volume data is usually stored in compressed form. So symmetry partition is not a good candidate because of its unbalance load and high disk I/O overhead. We use the multi-host partition scheme. All nodes are divided into several groups, and each group is host-slave partitioned. Since the the size of the volume image that needs to be processed could vary a lot and the computation times may also differ greatly for different subvolumes, the symmetry partition is not a good candidate because of its load imbalance problem and high disk I/O overhead.

5.1 Host-Slave Implementation

Several schemes are implemented to alleviate or eliminate the problems associated with the host-slave partition.

In order to discuss our partition scheme, we oversimplify the algorithm stated in section 1 as in Listing 1.

Step 1. *read slices.* (takes very long time)
Step 2. *compute C^3 data* (takes short time)
Step 3. *fit (3,3,3) functions* (takes long time)
Step 4. *compute the polygons* (takes longest time)
Step 5. *goto step 1*

Listing 1. The Oversimplified Algorithm

The best work division for host-slave partition is that the host does steps 1 and 2, and the slaves do steps 3 and 4. Step 2 takes very short time but requires lots of involved data; it requires 27 pixel information to compute the C^3 data of a pixel. So the host perform step 2 to avoid much communication overhead. The host sends the C^3 data of subcubes to slave nodes, and the slave nodes can do steps 3 and 4.

So the algorithms for host node and slave nodes are in Listings 2 and 3.

```
Step 1. reads slices
Step 2. computes C^3 data
Step 3. transfers subcubes to and receives polygons
        from slave nodes
```

Listing 2. The Host Node's Algorithm

```
Step 1. request and get subcubes from the host
Step 2. fit (3,3,3) functions.
Step 3. compute polygons
Step 4. send polygons to the host
```

Listing 3. The Slave Nodes' Algorithm

The host-slave partition usually incurs some problems as described in section 3.2. We employ several strategies to reduce the communication overhead, and eliminate the problems of idle host and idle slaves.

Three actions are taken to reduce the communication overhead.

1. All communications are asynchronous.
   It alternates between two buffers while sending any message. As long as the *send_message* is called, the node returns immediately and can begin to pack data into the other buffer. It waits only when the previous message sending is not completed.

2. Start communication early.
   When a slave node is almost finished processing subcubes, it sends a request to the host, so the slave can process newly arrived subcubes without any delay. This reduce the communication latency.

3. Reduce the number of transferred message
   It takes almost the same time to send one subcube or 10 subcubes. The host packs a couple rows of subcubes into one transmission packet. This has a two-fold effect. It not only reduces the number of transferred packets, but also reduce the total transmitted bytes. The shared boundary between subcubes is sent only once instead of four times. A pixel can be shared by 4 subcubes, and thus it will be sent four times if it sends one subcube per packet.

   The slave nodes store calculated polygons in local memory. They empty the polygons to the host when the polygons data accumulate to a certain level.

The program uses an interrupt handler to process communication requests. So, it can still hand out subcubes or receive polygons even when it is doing something. That is, the host always *listen* to slave nodes, and thus eliminate the idle slave node problem due to an unattending host. It is very tricky to implement interrupt driven software. From our experience, many hard to trace bugs, which only happen once in a while, are related to the critical section violations.

As to the problem of the idle host, the host program is designed to use its idle time to help processing subcubes. When the host is waiting for all subcubes of two slice to be transferred, it also processes additional subcubes. It still allows request interrupt to arrive while it is helping process a subcube, so the data transmission to a slave is not delayed. Mutual exclusion is carefully implemented. The host sends out large quantities of subcubes to each node at the beginning of a slice, and reduces the number of sent subcubes exponentially when the slice is almost done. This avoids long host idle time waiting for a slave to complete.

The algorithms for host node is in Listings 4. There is little change in the slave's algorithm.

```
Step 1. read slice S_i and compute its C^3 data
Step 2. starts the interrupt handler so the subcube
        transmission to slaves can happen
        in background. This step takes little
        time since it does not wait.
Step 3. read new slice S_{i+1} and compute its C^3 data
Step 4. wait for all subcubes of the "old" slice S_i
        to be transferred to slave nodes by the
        interrupt driven handler. The subcubes
        formed by the new slice S_{i+1} read at
        step 3 is transferred at the next iteration,
        and not the current iteration.
        Also help processing subcubes while waiting.
Step 5. go to step 3
```

Listing 4. The Host Node's Interrupt Driven Algorithm

5.2 Multi-Host Implementation

Multi-host partition is used to solve the bottleneck problems caused by insufficient host's computing

power. All nodes are divided into several groups in which there is one host and some slave nodes. Each group processes a subset of slices (Figure 6 (a)). When a group finishes its share of slices, the nodes in that group are re-assigned as slave nodes to other unfinished groups (Figure 6 (b)). The number of nodes assigned to each unfinished group is proportional to the unfinished work load of that group.

The following is the detailed implementation: Each group determine its share of slices. The boundary slices of every group are overlapped to provide the proper boundary condition. Each group processes its share independently. There is no synchronization between groups. When a group finish a slice, it broadcasts a message to other group hosts, so each group is aware of how much work is left in the other groups. When a group has only half a slice left, it broadcasts a message to avoid the re-assignment of nodes from other groups to it. When a group finishes its work, it sends the total amount of calculated polygons to a specific node, and also registers that it is done. Then all nodes in the finished group are re-assigned to other group according to the remaining work of the other groups. The host sends messages to other group hosts to inform them of his new slaves. The very last group host need to do some cleanup such as obtaining the total amount of generated polygons and released all nodes.

Although the multi-host partition has better performance over symmetry and single-host partitions, it is relatively difficult to implement because it could cause a deadlock because of communication delay. Intel iPSC is a message passing, not a shared memory, system and each message takes time to be delivered and received. For example, although a finished group has posted a "done" message, the message might not be processed in time by the other groups. So two finished groups might re-assign its group nodes to each other, and thus cause a deadlock. One solution to avoid such deadlocks is simply to discard newly assigned slave nodes when the group has finished its work. Those discarded slave nodes become orphans which have no host, and don't do any more work.

6 Results

The input for our initial experiments was 65 slices of 512×512 resolution images of a sphere with gray level increasing with the sphere's radius. The image data is generated by the group host on the fly. The results of Table 1-4 are obtained using the (1,1,1) approximation from C^0 data. Table 5 is obtained from

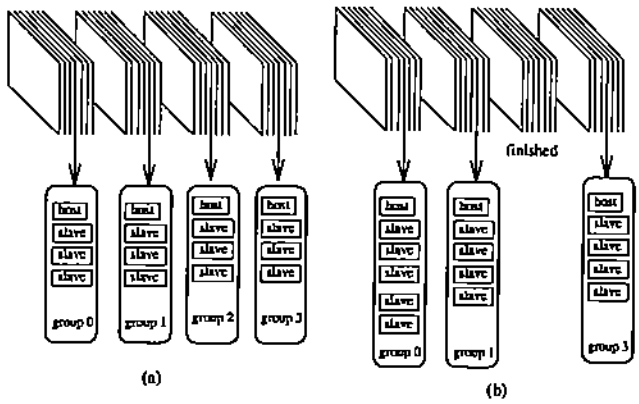

Figure 6: (a) Initial Multi-Host Partition (b) The Nodes of a finished Group are re-assigned to other Groups

by a (3,3,3) approximation of 65 slices with 192×128 resolution. The algorithm for (1,1,1) yielding a C^0 smooth approximation is very similar to the (3,3,3) approximation which is at least C^1 smooth. The output is seven sets of polygons, each set corresponding to a isosurface of a certain value. The total output size is 121,486 polygons for the $(1,1,1)$ approximation, and 78,741 polygons for the $(3,3,3)$ approximation. Each polygon have three to six vertices of XYZ coordinate values and a normal associated with each vertex.

The *total time* is the total processing time from the beginning to the end. It includes generating the sphere test data, message passing, processing and collecting calculated polygons. The polygons are transfered to the group hosts, and then discarded to avoid writing to disk. The *idle time* is measured as the time to wait for a message to come in. The *slave ave. idle* is the average slave idle time. The *sl. id. max − min* row shows the variation in slave idle time. The smaller number means better load balancing. The *speed/node* row shows the speed up per node. The maximum is 1. In the multi-host results Table 2,3 and 5, the *slave ave. idle* and *sl. id. max−min* are not available when there is more than one group. In multi-host partition, one node could be a group host and a slave at a different time. Or it could become an orphan which has no host (see the multi-host partition section), so it is hard to accurately measure the slave idle time.

Table 1 (at the end of the paper) shows the result of single host partition. The slave nodes begin to idle with the increasing number of nodes because they can not get subcubes fast enough to keep them busy. It is clear that the bottleneck occurs when the PE number is more than 8. The total processing time increases slightly with the increasing number of nodes because

the the extra slave nodes become a burden for the already busy host to take care of.

Table 2 shows the results of a multi-host partition scheme. Each group has eight nodes. When a group finishes its share, all nodes in that group are re-assigned to other group as slave nodes. The number of assigned nodes to each group is proportional to the remaining slices of that group.

Because the multi-host partition is a hybrid of host-slave and symmetry partitions, it has a little unbalance load problem when there are many groups. The unbalance load problem is alleviated, but not totally eliminated, by re-assigning nodes of finished group to unfinished groups. If the re-assignment is not performed, the "speed/node" ratio for 256 nodes drops from 0.564 to 0.481 as shown in table 2 and 3.

Table 3 shows the result if the finished group does not re-assign its nodes to other unfinished groups. It can be seen that the performance is worse than Table 2. The load balance between groups does not differ much because the test data is a perfect sphere. The degrading of Table 3 could become much worse if the processing time of different sections varies much.

The *speed/node* of 256 nodes degrades a lot because the slave idle time, during the host's reading two slices, begins to dominate. The slaves stay idle before its group host has read 2 slices inthe $(1,1,1)$ scheme or three slices in the $(3,3,3)$ approximation. When there are many groups, each group only process very limited number of slices. This idle time becomes a large portion of the total processing time. It is reasonable to preload the group host with two slices for the $(1,1,1)$ or three slices for the $(3,3,3)$ approximation. If we process hundreds of slices, this preloading time is negligible. Table 4 is obtained by preloading each host with two images before starting the time clock. This achieves near linear speed up.

Table 5 is obtained by the $(3,3,3)$ approximation obtained from 65 slices of 192×128 resolution (not the 512×512 for $(1,1,1)$). The calculation of $(3,3,3)$ takes a much longer time than $(1,1,1)$ with the same size of image. So the communication time becomes less important. The slave idle time is mainly caused by an unattending host and communication delay and is smaller compared to $(1,1,1)$. The bottleneck of the single host partition won't occurs when a host has fewer than 32 slave nodes. The *speed/node* of two nodes is relatively worse than four or eight nodes. This might be caused by the waiting time for the slave to fetch the last subcube of a slice. The host use its idle time to help process additional subcubes, but it won't finish the last subcube and start processing a new slice for programmatic reasons. So the host needs to be idle until the last subcube is requested by any slave. This idle time might be very long if a slave is given a large chunk of subcubes a time. When a slice is almost done, the host reduces the number of sent subcubes exponentially to a predefined minimum number to achieve a shorter host idle time. The expected idle time of this kind of the partition with only one slave should be N times that of the partition with N slaves if the slaves' requests are modeled as a Poisson process.

7 Conclusion

The results shown in Tables $1-5$ do not involve the writing of polygons to disks. We did experiment with the different partitions with the output files being written to disk. The slave idle becomes longer since writing files takes a large amount from the group hosts' processing time. It has better performance when one group has fewer number of nodes. The optimal number of nodes in a group is heavily dependent on the hosts' file I/O time and the slaves' processing time. Longer slaves' processing time or smaller I/O time means the optimal group size will be large. Our implementation provides one way to manually achieve near optimal performance.

Other features of our implementation include:

1. low I/O overhead

2. high efficiency even when a group has only a few nodes because the host help in processing subcubes

3. The host still pays attention to its slaves to achieve low slave idle time even when it is doing something else such as reading files or doing numerical calculations.

Having parallel MIMD algorithms for modeling and visualization is also only a step way from distributed computations on a network cluster of workstations (with slightly different tradeoffs). In [1] we present details of a distributed volume rendering computation of volume images based on distributed ray casting on a network cluster. We are also now porting our approximation and iso-contouring parallel MIMD modeling and visualization algorithms to a network cluster.

References

[1] V. Anupam, C. Bajaj, D. Schikore, and M. Schikore. *Distributed and Collaborative Volume Visualization. IEEE Computer*, x(x):x–y, 1994.

[2] C. Bajaj and G. Xu. *Trivariate Interpolants and Scientific Visulization.* Computer Science Technical Report, CSD-TR-93-18, Purdue University, 1993.

[3] T. Crockett and T. Orloff. A mimd rendering algorithm for distributed memory architectures. In Proceedings of the IEEE/ACM 1993 Parallel Rendering Symposium, pages 35–42, 1993.

[4] D. Ellsworth. A multicomputer polygon rendering algorithm for interactive applications. In Proceedings of the IEEE/ACM 1993 Parallel Rendering Symposium, pages 43–48, 1993.

[5] Lorensen, W., and Cline, H. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21:163–169, 1987.

[6] S. Whitman. A task adaptive parallel graphics renderer. In Proceedings of the IEEE/ACM 1993 Parallel Rendering Symposium, pages 27–34, 1993.

[7] J. Wilhelms and A. Van Gelder. Octrees for Faster Isosurface Generation. *Computer Graphics*, 24:57–62, 1990.

# of nodes	128	64	32	16	8	4	2	1
total time (sec)	107.8	99.1	93.8	90.7	143	284	567	1110
host idle time	0.3%	0.2%	0.1%	0.1%	0.0%	0.1%	0.5%	0%
slave ave. idle	91.2%	82.3%	62.9%	22.3%	1.7%	0.8%	0.4%	N/A
sl. id. max-min	0.7%	0.9%	0.4%	0.5%	0.0%	0.0%	N/A	N/A
speed/node	0.080	0.170	0.370	0.756	0.964	0.976	0.978	1.0

Table 1. The result of single host partitions.

# of nodes	256	128	64	32	16	8	4	2	1
total time (sec)	7.69	12.06	20.90	38.8	73.8	143	284	567	1110
host idle time	0.3%	0.3%	0.1%	0.0%	0.0%	0.0%	0.1%	0.5%	0%
slave ave. idle	N/A	N/A	N/A	N/A	N/A	1.7%	0.8%	0.4%	N/A
sl. id. max-min	N/A	N/A	N/A	N/A	N/A	0.0%	0.0%	N/A	N/A
speed/node	0.564	0.719	0.830	0.894	0.940	0.964	0.976	0.978	1.0

Table 2. The result of multi-host partitions, where each group has 8 nodes.

# of nodes	256	128	64	32	16	8	4	2	1
total time (sec)	9.02	13.39	22.23	40.3	76.5	143	284	567	1110
host idle time	0.5%	0.2%	0.2%	0.2%	0.1%	0.0%	0.1%	0.5%	0%
slave ave. idle	N/A	N/A	N/A	N/A	N/A	1.7%	0.8%	0.4%	N/A
sl. id. max-min	N/A	N/A	N/A	N/A	N/A	0.0%	0.0%	N/A	N/A
speed/node	0.481	0.648	0.780	0.862	0.908	0.964	0.976	0.978	1.0

Table 3. The result of multi-host partitions with no node re-assignment.

# of nodes	256	128	64	32	16	8	4	2	1
total time (sec)	5.237	9.60	18.45	36.36	71.38	141	282	565	1108
speed/node	0.826	0.902	0.938	0.952	0.970	0.979	0.982	0.983	1.0

Table 4. The result of multi-host partitions if each group is preloaded with two data slices.

# of nodes	256	128	64	32	16	8	4	2	1
total time (sec)	6.20	10.7	18.8	37.6	76.4	156	318	693	1165
host idle time	3.1%	1.9%	1.5%	0.7%	0.6%	0.3%	1.2%	4.9%	0%
slave ave. idle	N/A	N/A	N/A	N/A	1.4%	2.4%	0.7%	0.2%	N/A
sl. id. max-min	N/A	N/A	N/A	N/A	2.3%	1.8%	0.1%	N/A	N/A
speed/node	0.733	0.849	0.966	0.967	0.953	0.934	0.917	0.840	1.0

Table 5. The result of multi-host partitions for $(3, 3, 3)$ *approximation.*