Parallel Manipulations of Octrees and Quadtrees*

Vipin Chaudhary[1], K. Kamath[2], P. Arunachalam[3], and J. K. Aggarwal[3]

[1] Department of Electrical and Computer Engineering
Wayne State University, Detroit, MI 48202, USA
[2] Sun Microsystems, San Jose, CA 95134, USA
[3] Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, TX 78712, USA

Abstract. Octrees offer a powerful means for representing and manipulating 3-D objects. This paper presents an implementation of octree manipulations using a new approach on a shared memory architecture. Octrees are hierarchical data structures used to model 3-D objects. The manipulation of these data structures involves performing independent computations on each node of the octree. Octrees are much easier to deal with than other forms of representations used to model 3-D objects especially where extensive manipulations are involved. When these operations are distributed among multiple processing elements (PEs) and executed simultaneously, a significant speedup may be achieved. Manipulations such as a complement, a union, an intersection and other operations such as finding the volume and centroid which this paper describes are implemented on the Sequent Balance multiprocessor. In this approach the PEs are allocated dynamically, resulting in a uniform load balancing among them. The experimental results presented illustrate the feasibility of the approach. Although this evaluation has been originally done for shared memory machines, it will provide insight for the evaluation on other architectures.

1 Introduction

Efficient manipulations of 3-D objects are important in various applications such as computer graphics, computer vision and other related areas. These schemes can be categorized as surface descriptions or volumetric descriptions. Chien and Aggarwal [1] summarize advantages and disadvantages of each category. Most representation techniques suffer from severe memory and processing requirements with increasing input requirements [2]. The octree is a well known data structure in the representation of 3-D objects. It is used to determine various geometric properties such as volume and centroid and to manipulate objects by computing their complement, union, and intersection. An octree representation scheme uses efficient tree traversal algorithms to overcome the drawbacks stated earlier. Chen and Huang [3] survey in detail the construction of octrees. Though

* This research was supported in part by IBM.

this can be done on a sequential machine, the nature of the algorithm suggests the use of a parallel machine. As these algorithms use three orthogonal views to generate the octree, they may pose problems for objects with cavities as three views are insufficient to generate an exact 3-D description of an object with cavities. Chien and Aggarwal [2] elaborate on an octree generation from more than three views. While a tree structure indicates an increase in the data dependencies, the regularity of the structure presents ways to avoid this problem. Samet [8] presents a detailed study of the complexity of the tree traversal algorithms. Moitra and Iyengar [9] also discuss an idea of the parallelism which can be found in such algorithms.

The octree structure for the representation of 3-D objects is an extension of the quadtree structure for 2-D objects. The octree manipulation is computationally expensive because of the huge volume of data. Hence, it makes sense to parallelize the operations especially in real time systems. However, due to the tree nature of the algorithm the parallelization is not easy and requires more complicated techniques. Another problem with these algorithms is that they are not computationally intensive and require more data communication than inherent computation on a single node. Due to this reason the speedup increase is not linear with the increase in number of PEs.

The rest of the paper is organized as follows. The next section describes the parallel algorithms for generating octrees from three orthogonal views. Section three describes the manipulation of octrees involving the union and intersection of two octrees, evaluating the volume and centroid of an octree, and finally the displaying of the octree as an object. The results of the implementation of the above algorithms on the Sequent Balance multiprocessor are detailed in section four. We conclude in section five with comments on the results of our implementation and possible extensions of this work.

2 Representation of Octrees

The hierarchical representation of an octree represents a binary image in a compressed form. Computations to be performed on these data structures can be considered as simple tree traversal algorithms which can be efficiently implemented in parallel. It is assumed that the objects are specified in a binary format with an image represented by white pixels and background by black pixels. The figures representing the objects have been drawn in the inverse format. Figure 1 illustrates an example of an octree and its three orthogonal views.

2.1 Parallel Method for Octree Construction

The octree of a binary image is constructed by subdividing the image into eight octants recursively until each octant is either fully white (object) or fully black (non-object). Each octant is a node in the tree, and each node can be a terminal node (leaf node) or a non-terminal node (grey node). A leaf node can be white or black and a grey node which is non-terminal, is a subtree which defines a part of

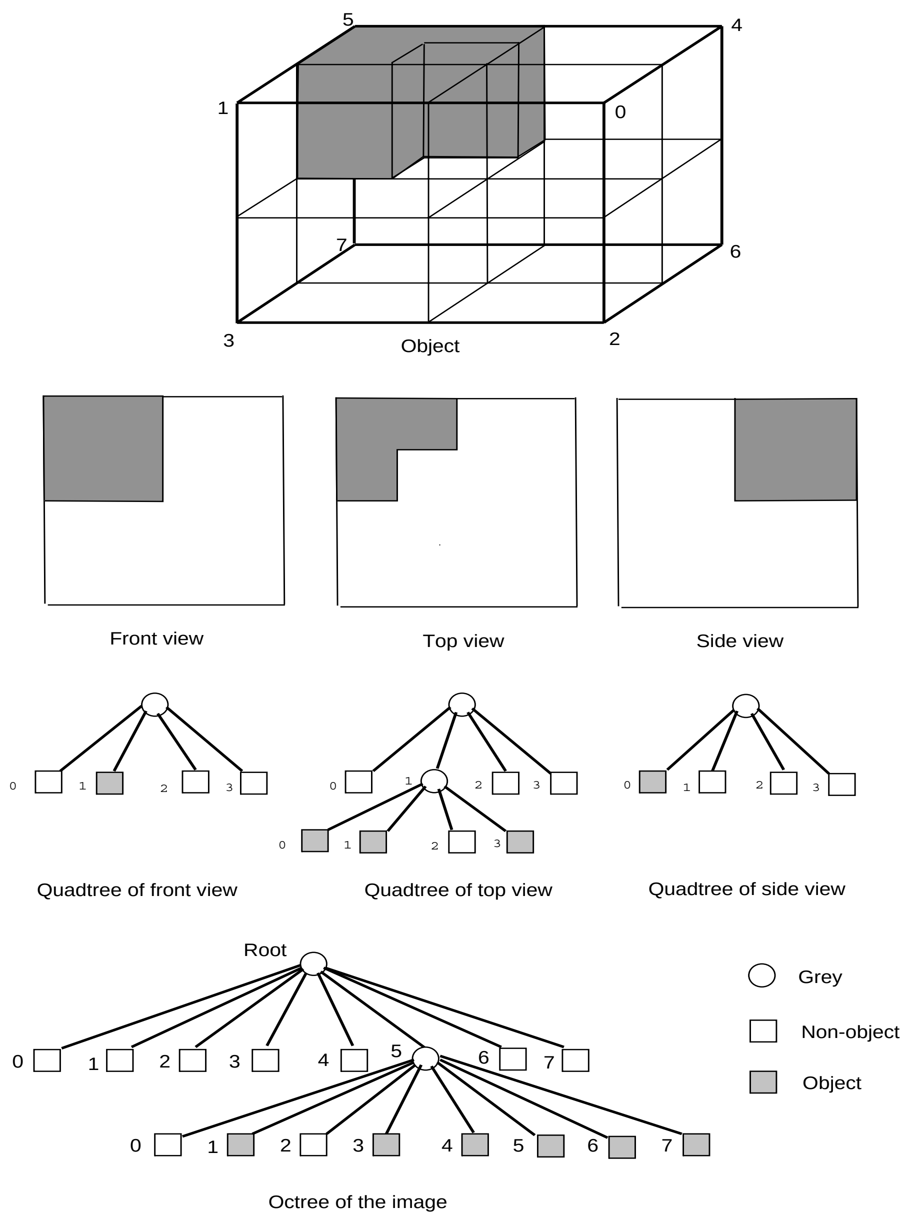

Fig. 1. Example of an octree generation by three orthogonal views of the object

the object neither completely white nor black. Each node in an octree contains information regarding the structure of the octree. This information includes the color, the surface pointers, the children, and the node pointers. Three quadtrees corresponding to the top, front, and the side view of the object are first constructed from the scans of the three images. These are then intersected to get the final octree of the object. Hence, the quadtree generation algorithm is used thrice to get the quadtrees of the individual views, and the octree is obtained from this. Table 1 gives the various octants of an image depending on the view direction and the position of the quadrant.

Table 1. Illustration of the various octants of an image depending on the view direction and the position of the quadrant.

View	NE	NW	SE	SW
Front	0, 4	1, 5	2, 6	3, 7
Top	4, 6	5, 7	0, 2	1, 3
Side	4, 5	0, 1	6, 7	2, 3

This part presents the parallel algorithm used for generating an octree. The three quadtrees are traversed in parallel, and a logical AND operation is performed on the node pairs. This intersection table data is maintained in the shared memory. A task queue is set up, and the root pointers of the three quadtrees and the octree pointer are inserted as the task in the task queue. When the algorithm is invoked, a check is performed to see if all the three octnodes (octree nodes) have the same color. If not, eight children corresponding to the eight octants are created and appended to the task queue which are obtained from the children of the various quadtrees. This process continues until all the three quadtree nodes checked are of the same color. An idle PE (Processing Element) picks up the next available task in the task queue and executes it. The entire job is complete when the task queue is empty and the entire image has been created. It should be noted that an empty queue does not imply a completion of the job because the task may not have been appended to the queue. Table 2 gives the intersection table between the quadtrees and the final octree.

2.2 The Parallel Algorithm for Octree Generation

This section presents (more elaborately) the actual algorithm used is the study. An idle PE takes the task from the task queue and performs the following operation:

1. If all the three quadtree nodes are grey, then the octnode is marked grey, eight child nodes are appended to it and eight tasks are created in the task queue. Each task contains a child node of the octree and its corresponding three intersection quadtree nodes referred from the intersection table.

Table 2. Intersection table between the quadtrees corresponding to the final octree.

Quadtree node of front view	Quadtree node of top view	Quadtree node of side view	Corresponding octree node
0	2	1	0
1	3	1	1
2	2	3	2
3	3	3	3
0	0	0	4
1	1	0	5
2	0	2	6
3	1	2	7

2. If all the three quadtree nodes are black, then the corresponding octree node is marked black.
3. If one of the quadtree nodes is marked white, then the corresponding octree node is marked white.
4. If two of the quadtree nodes are marked grey and the third is black or null, then the corresponding octree node is marked grey. Then, eight nodes are added to the octnode, and eight tasks are appended to the task queue. Each entry in the task queue has two pointers to the children of the nodes marked grey, and the third is marked null.
5. If one of the quadtree nodes is grey, one is black, and the third is null, then the corresponding octree node is marked grey. Then, eight nodes are added to the octnode, and eight tasks are appended to the task queue. Each entry in the task queue has a pointer to a child node of the grey node while the other two pointers are marked null.
6. If two of the nodes are marked black and the third is marked null, then the corresponding octree node is marked black.
7. If two of the nodes are marked white and the third is marked black, then the corresponding octree node is marked white.

In algorithm 1,

1. the total object and background pixels are assumed to be a cube of dimension $bsize$.
2. q_ptr_1, q_ptr_2, and q_ptr_3 represent the pointers of the corresponding quadtrees for the octree.
3. q_size_1 represents the size of the first quadtree.
4. T_const(meaning Tree constant) is a constant dependent on the image being a quadtree or an octree. This is 4 for a quadtree and 8 for an octree.

```
Algorithm 1. Octree Generation.

begin
  while(IMAGE_SIZE≠ bsize^3)
      while(TASK_QUEUE≠ Empty)
          if(q_ptr1 = q_ptr2 = q_ptr3 = GREY)
            O_ptr = O_ptr + T_const
            O_color = GREY
            TASK_QUEUE = TASK_QUEUE + T_const
          else if(q_ptr1 = q_ptr2 = q_ptr3 = BLACK)
            O_color = BLACK
            B_SIZE = q_size1
          else if(q_ptr1 or q_ptr2 or q_ptr3 = WHITE)
            O_color = WHITE
            B_SIZE = q_size1
          else if(two q_ptrs are GREY, third q_ptr is BLACK or NULL)
            O_ptr = O_ptr + T_const
            O_color = GREY
            TASK_QUEUE = TASK_QUEUE + T_const and q_ptr3 = NULL
          else if(one q_ptr is BLACK, one q_ptr is GREY, one q_ptr is NULL)
            O_ptr = O_ptr + T_const
            O_color = GREY
            TASK_QUEUE = TASK_QUEUE + T_const
          else if(two q_ptrs are BLACK and the third is NULL)
            O_color = BLACK
            B_SIZE = q_size
          else if(one of the two q_ptrs is WHITE, and the third is NULL)
            O_color = WHITE
            B_SIZE = q_size
  IMAGE_SIZE = IMAGE_SIZE+ B_SIZE
end
```

Each PE executes the above code concurrently. In the implementation on a shared memory machine, all the shared variables are locked when they are updated to prevent simultaneous accesses by many processors and thus to avoid erroneous values. The entire task queue, tail, and image size are the variables shared by the entire process whereas all other variables are local to a processor. All locked variables are accessible by only one processor; the other processors must wait until it is released. This retards the speedup achieved by parallel processing with the shared memory paradigm.

To get a picture of the object, one needs to store surface information explicitly in the octree nodes. Using this, a 2-D shaded image of the object can be obtained. Such an octree is called a volume/surface (VS) octree. This is usually done using a *multi level boundary scan*. This scheme was first suggested by Chien and Aggarwal [1] and used to detect all the interfaces between the object and the surrounding volume. As no neighbor finding operations are involved, the implementation is easier and faster. This is similar to that of Jackins and Tanimoto [6] and more generalized than that which was suggested by Doctor and Torborg [7].

3 Manipulation of Octrees

A wide variety of information can be obtained from the octrees. Such information as evaluating the volume and centroid of the object, getting 2-D projections of the object from various views and angles, and finding the complement, the union, and the intersection of the object can be obtained.

3.1 Union and Intersection of Octrees

The union and the intersection of octrees also involve the tree traversal concepts. Figure 2 illustrates the union and Fig. 3 illustrates the intersection of two objects (and the corresponding octrees) respectively. It can be seen that the resulting octree is obtained by manipulating the two octrees of the individual objects themselves. The intersection involves traversing the trees in parallel and performing a logical AND operation between them. The logical AND is necessary as it generates a 0 if either object is absent and generates a 1 if both are present. As the union of objects implies that the final image should have a pixel at any point if either of the two objects are present at that location, the equivalent logic operation is used. The union involves performing a logical OR operation between the two trees.

Algorithm 2. Union of octrees

begin
  while(IMAGE_SIZE$\neq$ $bsize^3$)
    while(TASK_QUEUE$\neq$ $Empty$)
      if(o_ptr_1 = o_ptr_2 =BLACK)
        U_color = BLACK
        B_SIZE = min(o_ptr_1.size, o_ptr_2.size)
      else if(o_ptr_1 or o_ptr_2 = WHITE
        U_color = WHITE
        B_SIZE = min(o_ptr_1.size, o_ptr_2.size)
      else
        U_ptr = U_ptr + T_const
        U_color = GREY
        TASK_QUEUE = TASK_QUEUE + T_const
  IMAGE_SIZE = IMAGE_SIZE+ B_SIZE
end

In algorithm 2 and 3,

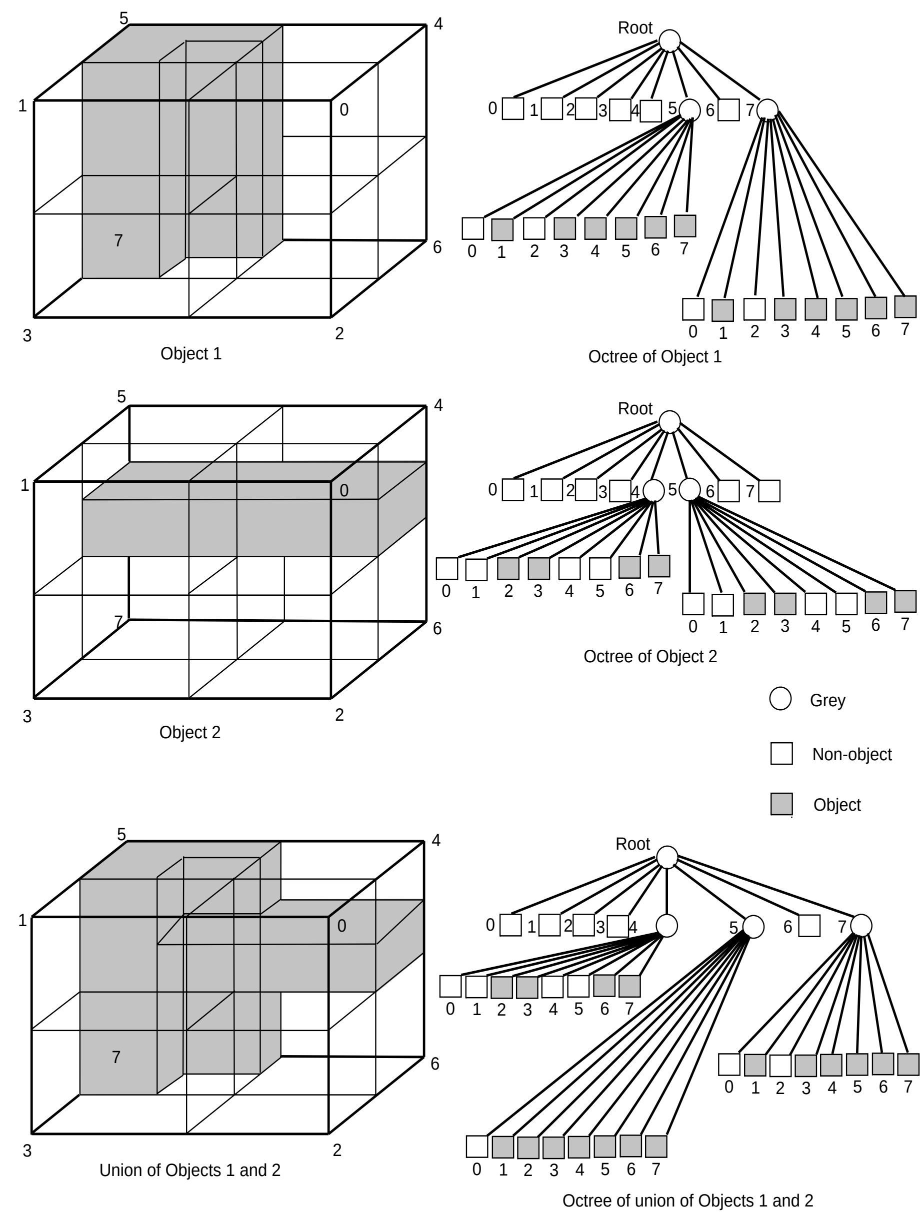

Fig. 2. Union of two objects and the corresponding octrees

1. the total object and background pixels are assumed to be a cube of dimension $bsize$;
2. o_ptr_1 and o_ptr_2 are the pointers of the two octrees whose union is sought; and
3. o_ptr_1.size and o_ptr_2.size represent the sizes of the nodes of the octree.
4. U_ptr and U_color are the pointers to and the colors of the union respectively.

In intersection, the root nodes of both the octrees and the intersection octree are inserted as a task in the task queue. An idle PE takes the task from the task queue and starts executing using the algorithm stored in the local memory. If the node pointed to by q_ptr$_1$ and q_ptr$_2$ is white, the resultant node in the intersection is marked as white. If one of the nodes is black, the resultant node in the intersection is marked as black. If both are grey, or if one is grey and the other is white, the resultant node in the intersection is marked as grey. Now eight child nodes are created and are appended to the queue. Many PEs can take these tasks and process them independantly resulting in an increased speed of operation. The operation of the union is similar. If the node pointed to by q_ptr$_1$ and q_ptr$_2$ is black, the resultant node in the union is marked black. If one of the nodes is white, the resultant node in the union is marked white. If both are grey, or if one is grey and the other is black, the resultant node in the intersection is marked grey. Now eight child nodes are created and are appended to the queue.

Algorithm 3.Intersection of octrees

```
begin
   while(IMAGE_SIZE≠ bsize^3)
        while(TASK_QUEUE≠ Empty)
             if(o_ptr1 = o_ptr2 =WHITE)
               U_color = WHITE
               B_SIZE = min(o_ptr1.size, o_ptr2.size)
             else if(o_ptr1 or o_ptr2 = BLACK
               U_color = BLACK
               B_SIZE = min(o_ptr1.size, o_ptr2.size)
             else
               U_ptr = U_ptr + T_const
               U_color = GREY
               TASK_QUEUE = TASK_QUEUE + T_const
   IMAGE_SIZE = IMAGE_SIZE+ B_SIZE
end
```

3.2 Displaying the Object

In order to display the octree as an object, an operation needs to be performed which will detect the boundary surfaces of the object. Meagher [5] first suggested

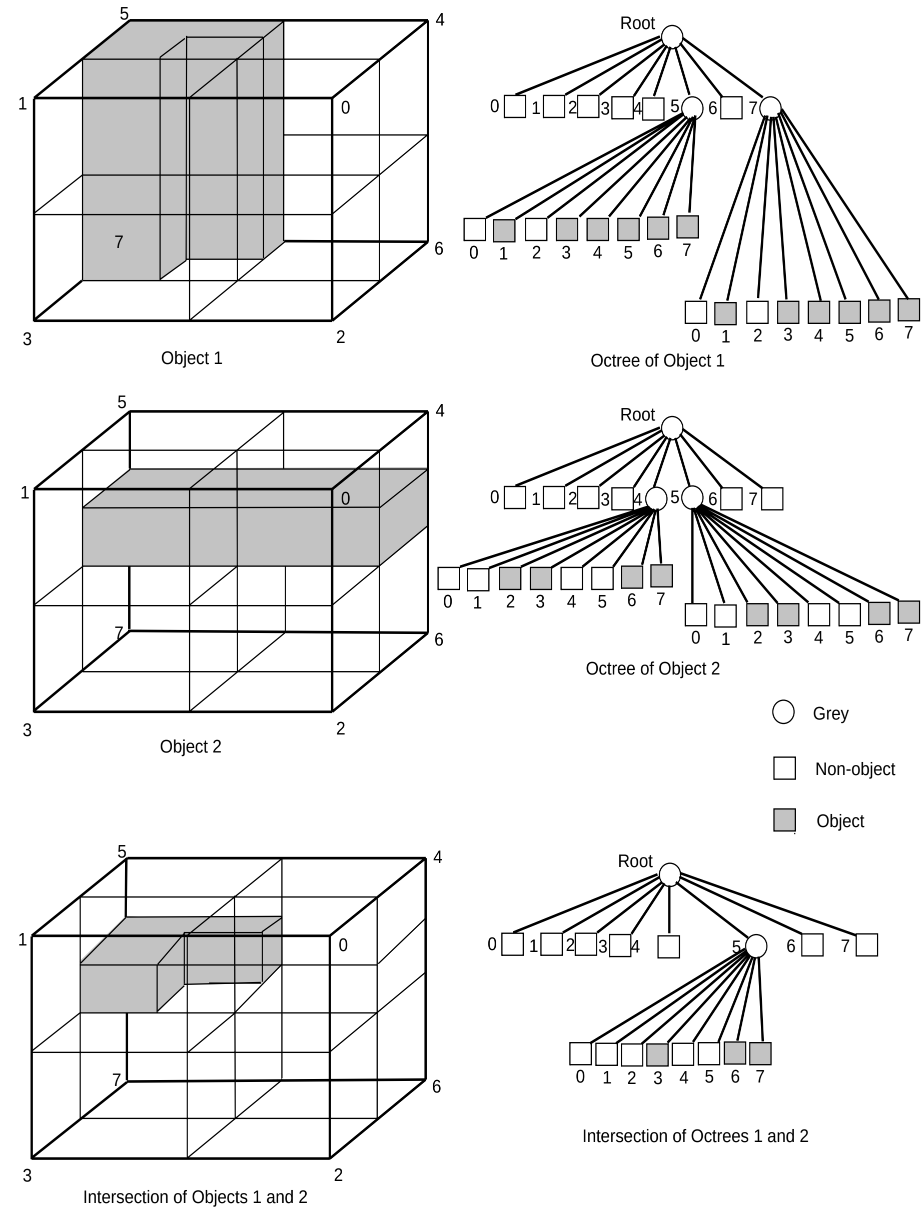

Fig. 3. Intersection of two objects and the corresponding octrees.

attaching the surface normals to the nodes of the octrees. An octree can have the surface information stored explicitly in its nodes. Such an octree is called a volume/surface (VS) octree. The surface information is computed using the *Multi level Boundary Search* (MLBS) method. Here, the octree is traversed, and the surface normals of the surface nodes are computed from the adjacency information. The orientation of the surface normals is coarsely quantized into 26 directions and stored as the surface information in the nodes.

In 3-D space there are 12 interfaces for 12 different combinations of the child node pairs that are adjacent to each other. The orientation of the surface normals is maintained in a table. The root node of the octree is inserted as a task into the task queue. An idle PE picks up the task and refers to the adjacency table to find the nodes adjacent to the node picked from the queue. For each of those adjacency nodes the following operations are performed.

1. If both of the octree nodes are grey, then the four pairs of child nodes adjacent to each other are appended to the task queue.

2. If one of the octnodes is black and the other is white, then the surface information is stored in the black node.

3. If one octnode is grey and the other non-grey, then the child nodes of the grey node and the non-grey node are appended to the task queue.

As the positions of the black and white nodes are known, the surface normals of each block can be computed by averaging the directions of all black and white interfaces of each block. The directions are assumed to be moving from the white to the black (object to surounding). The following is the algorithm for this operation.

Algorithm 4. Multi-level boundary search.

begin
  while(IMAGE_SIZE$\neq bsize^3$)
    while(TASK_QUEUE$\neq Empty$)
      if(o_ptr_1 = GREY and o_ptr_2 = NULL)
        TASK_QUEUE = TASK_QUEUE +20
        $\text{TASK}_{o\_ptr_1} = \text{TASK}_{o\_ptr_1} + \text{T\_const}$
        $\text{TASK}_{o\_ptr_1} = \text{TASK}_{o\_ptr_1} + 12$ for each adjacent pair
      else if($o\_ptr_1 = o\_ptr_2$ = GREY)
        TASK_QUEUE = TASK_QUEUE +4
        $\text{TASK}_{o\_ptr_2} = \text{TASK}_{o\_ptr_2} + 4$ for pairs adjacent to o_ptr_1
        $\text{TASK}_{o\_ptr_1} = \text{TASK}_{o\_ptr_1} + 4$ for pairs adjacent to o_ptr_2
      else if(o_ptr_1 =BLACK and o_ptr_2 =WHITE)
        o_ptr_{BLACK} = surface
        B_SIZE = min(o_ptr_1.size , o_ptr_2.size)
      else if(o_ptr_1 =GREY and $o\_ptr_2 \neq$ GREY)
        TASK_QUEUE = TASK_QUEUE +4
        $\text{TASK}_{o\_ptr_1} = \text{TASK}_{o\_ptr_1} + 12$ for each adjacent pair
  IMAGE_SIZE = IMAGE_SIZE + B_SIZE
end

1. the TASK_QUEUE is the queue made for the elements of the search.
2. the TASK_{o_ptr1} implies those tasks of the first octree.
3. the o_ptr_{BLACK} are those octree pointers which point to octnodes which are BLACK.
4. surface implies that the octree pointer referred to is on the surface.

Using this method, a 3-D object can be given a 2-D projection. The visibility of each block is determined by the dot product of its surface normal and the viewing direction. Since the surface directions are quantized, a set of surface directions visible are obtained by taking a dot product of the viewing direction and the 26 orientations of surface normals. Once the MLBS is carried out, the surface information is stored in each black terminal node. To get the projection, the root node is inserted in the task queue. An idle PE takes up this task and starts executing it. If the color is grey, eight tasks are appended to the task queue, and the tree is traversed to a lower level. If the octree node pointed to is black, its surface normal is compared with the set of visible orientations. If there is a match, the node is projected onto the screen.

3.3 Other Operations on Octrees

The volume of the object can be calculated easily from the octree by traversing down the tree until every white node has been visited. From the level of the node, the volume of each individual octnode can be computed and summed up to get the total volume of the object. If n is the level of the octnode, then 2^{3*n} is the volume of the octnode.

The centroid of the object is a point which is the average of all the white pixels in that coordinate. To locate the centroid the octree must have a structure which includes the starting points of the three coordinates. A procedure can be constructed which computes the centroid of the object by summing up the products of the centroid and the volume of each node, and then dividing this sum by the total volume. If X_{cent}, Y_{cent} and Z_{cent} are the coordinates of the centroid, the centroids of the individual nodes can be easily obtained by using the formula $X_{cent} = X_{start} + 2^{n-1}$ where X_{start} is the coordinate of the starting address of the node, and similarly for Y_{cent} and Z_{cent}. Thus the final centroid is given by

$$X_{cent} = \frac{1}{Vol} \sum_{i=1}^{n} X_{cent_i} * Vol_i,$$

$$Y_{cent} = \frac{1}{Vol} \sum_{i=1}^{n} Y_{cent_i} * Vol_i,$$

$$Z_{cent} = \frac{1}{Vol} \sum_{i=1}^{n} Z_{cent_i} * Vol_i$$

where n is the total number of octnodes, and Vol is the total volume.

The complement of an image is obtained by changing all the white pixels to black and black pixels to white. This is accomplished in octrees by creating a complimentary tree and inserting it in the task queue. Any idle PE starts on the octree and traverses down. If the node is marked black, the node in the complementary tree is marked white. If the node is marked white, the node in the complementary tree is marked black. However, if the node is marked grey, eight child pointers are appended to the task queue, and eight nodes are created in the *c_tree* where *c_tree* is the complementary tree. Idle PEs will pick up tasks from the task queue and execute them as long as they are available in the task queue. The algorithm terminates when the task queue is empty, and the volume of the complementary image equals that of the original image.

4 Results

All the algorithms were implemented on the Sequent Balance multiprocessor. We used images of two different sizes for timing analysis. The results of these implementations are shown in Fig. 4. The display was done using the multi-level boundary search. The speedups obtained for images of different sizes are shown separately. It should be mentioned that the amount of computation on each node is much smaller than the amount of time required to create child processes on the Sequent Balance and inserting them into the task queue. Thus, a large

Fig. 4. Results of the various implementations: (a) Octree generation, (b) Union of Octrees, and (c) Intersection of Octrees.

portion of the time is spent in system overhead and not actual processing. This is an inherent drawback of the load balancing paradigm we follow and was found quite visible with lesser image sizes. Examples of intersection and union of some objects are are also shown.

It can be seen that while there is a great variation in the speedups, general trends can be noted. In general, the times taken for smaller images are lesser than those for larger images, provided the images are complicated enough. If the images are simple, then time required for the octree construction would be obviously less. The time necessary for union or intersection of octrees is much more than that of a single octree generation as three octrees need to be constructed (two for the initial objects and one for the final object) with a large increase in the amount of shared data and thus problems with variable locking. It is also seen that the time taken increases after a certain number of PEs. This is mainly due to the increase in the overhead of creating new tasks and the shared data being much larger in volume than the data local to a processor.

We verified the algorithms by displaying the union and intersection of several objects. Figures 5, 6, 7, and 8 verify the correctness of the algorithms.

Fig. 5. In clockwise order from top left: Object 1; Object 2; Union of objects 1 and 2; Intersection of objects 1 and 2.

5 Conclusions

It can be inferred from the graphs that low speedups were obtained for smaller image sizes but the speedups are generally low even in a large image size. This

Fig. 6. In clockwise order from top left: Object 1; Object 2; Union of objects 1 and 2; Intersection of objects 1 and 2.

Fig. 7. In clockwise order from top left: Object 1; Object 2; Union of objects 1 and 2; Intersection of objects 1 and 2.

Fig. 8. In clockwise order from top left: Object 1; Object 2; Union of objects 1 and 2; Intersection of objects 1 and 2.

is due to the large overhead involved in the creation of new tasks (50ms for each). While dynamic scheduling may improve the load balancing, it does create other processes which actually increase the load. The bottleneck of the entire computation is the shared task queue whose access is mutually exclusive to the processors. Since the task sizes in all the octree algorithms are extremely small, most of the processors are waiting for access to the globally shared task queue. Thus, there is a tradeoff involved between load balancing and speedup.

It will be worthwhile to compare the speedups obtained by statically partitioning the data or by using a combination of static and dynamic partitioning.

6 Acknowledgements

It is our pleasure to acknowledge the help of Mr. Michael Schulte in debugging a part of the code used for implementation.

References

1. Chien, C. H., Aggarwal, J. K.: Volume/surface octrees for the representation of three-dimensional objects. Computer Vision, Graphics and Image Processing. **36** (1986) 100–113
2. Chien, C. H., Aggarwal, J. K.: Reconstruction and matching of 3D objects using quadtrees/octrees. Proceedings of 3rd Workshop on Computer Vision. (1985) 49–54

3. Chen, H. H., Huang, T. S.: A survey of the construction and manipulation of octrees. Computer Vision, Graphics and Image Processing. **43** (1988) 409–431
4. Schneier, M.: Calculations of geometric properties using quadtrees. Computer Graphics and Image Processing. **16** (1981) 296–302
5. Meagher, D. J. R.: The octree encoding method for efficient solid modeling. Ph. D. dissertation, Electrical and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, New York 12181.
6. Jackins, C. L., Tanimoto, S. L.: Quad-trees, oct-trees and K-trees: a generalized approach to recursive decomposition of euclidean space. IEEE Trans. Pattern Anal. Mach. Intell. **PAMI-5** (1983) 533–539
7. Doctor, L. J., Torborg, J. G.: Display techniques for octree-encoded objects. IEEE Comput. Graphics Appl. **3** (1981) 29–40
8. Samet, H.: A top-down quadtree traversal algorithm. IEEE Trans. Pattern anal. Mach. Intell. **PAMI-7** (1985) 94–98
9. Moitra, A., Iyengar, S. S.: Parallelism from recursive programs. Advances in Computers. **June** (1986)