A PARALLEL ALGORITHM FOR BINARY SPACE PARTITIONING TREES

BY
NAKWON CHU

Bachelor of Engineering
Inha University
Inchen, Korea
1984

Master of Science
Inha University
Inchen, Korea
1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1996

A PARALLEL ALGORITHM FOR BINARY SPACE PARTITIONING TREES

Thesis Approved:

Thesis Adviser

H. Lu

J. Chandler

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. K. M. George, my advisor. I appreciate not only his time spent on my thesis but also his encouragement and advice throughout my graduate study. I would like to extend my special thanks to Dr. J. P. Chandler for his invaluable suggestions and comments. A special thanks and appreciation to Dr. H. Lu for her suggestions and help.

To all my family, but especially my parents, Young-Taek Chu and Sun-Soon Mo, my heartfelt thanks for all the many years of love and support. My thanks also go to my brother-in-law and sister for their assistance. And finally, I wish to express my thanks to my friend, Min-Su Choi and Se-Jin Choi, for their time and experience in reviewing and proofing my programs.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	REVIEW OF LITERATURE	3
	Hidden Surface Elimination Algorithms	3
	Ray Tracing	3
	Painter's Algorithm	4
	Z-Buffering Algorithm	5
	BSP Tree Algorithm	5
	Comparison of Algorithms	6
	Parallel Processing Model	7
III.	BSP TREE OPERATION	10
	Building BSP Tree	10
	Traversing BSP Tree	14
IV.	PARALLEL PROGRAMMING MODEL FOR BSP TREE CONSTRUCTION	16
	Sequential Algorithm for Tree Construction and Display	16
	Parallel Algorithm for Tree Construction	18
V.	PERFORMANCE ANALYSIS	25
	Theoretical Analysis	26
	Experimental Analysis	29
VI.	CONCLUSION	34
	REFERENCES	36
	APPENDICES	
	APPENDIX A - ORGANIZATION OF THE PROGRAM	38
	APPENDIX B - DERIVATION OF A FORMULA	40

LIST OF TABLES

Table Page

1. Process to Processor Mapping........................ 21

2. Tree level and Number of Active Processors.......... 28

3. Execution Time, CPU Time, and efficiency of 500 Polygons.................................... 31

4. Execution Time, CPU Time, and efficiency of 1000 Polygons.................................... 32

5. Execution Time, CPU Time, and efficiency of 2000 Polygons.................................... 32

6. Execution Time, CPU Time, and efficiency of 5000 Polygons.................................... 33

LIST OF FIGURES

Figure Page

1. BSP Tree Construction.................................. 11

2. Process Decomposition of Build-Tree Algorithm........ 20

3. Mapping of n Processors.............................. 22

4. Polygon Shape.. 25

5. Polygon Set After Back To Front Sorting.............. 26

LIST OF ALGORITHMS

Algorithm Page

1. Sequential BSP-Tree-Build Algorithm.................. 16

2. BSP-Tree-Display Algorithm........................... 18

3. Parallel BSP-Tree-Build Algorithm.................... 24

CHAPTER I

INTRODUCTION

One of the long term goals of computer graphics is real-time generation of realistic images of simulated 2-D and 3-D environments. Ten years ago, creating an image in 1/60 of a second, fast enough to continually generate images on a video monitor, is considered `Real-time'[1]. In 1993 image creating speed was found to be nine or ten times faster than ten years ago [24]. Currently the hardware is even faster. With this fast image generation, there is no discernible delay between specifying parameters for an image and the image's appearance on the monitor's screen.

There are two major approaches to get real-time performance (fast speed): Brute force and parallelism [19]. Brute force method employs larger and faster computers, requiring larger budgets. Parallelism, on the other hand, often makes use of software technology to achieve the speed-up as fast image generation methods. But systems which can achieve high performance are currently very expensive. So, there are many algorithms being developed.

Binary Space Partitioning(BSP) is one of the algorithms to rapidly generate realistic images of 2-D and 3-D scenes composed of polygons. The BSP algorithm is used

for solving the hidden surface problem. The BSP algorithm is based on generating a `Binary Space Partitioning Tree.' An in-order traversal of a BSP tree at run-time will produce a linear order of visibility of polygons in relation to the viewing position. This visibility information can be used to speed up image generation.

The polygon sorting problem is the task of deciding the position of polygons on the display screen. This sorting problem is to determine which polygons lie behind which, as seen from the view point. When polygons overlap, we need to decide the order in which the polygons should be drawn.

The BSP tree algorithm partitions the space into two subspaces by a plane. The two sides of the plane are called `inside' and `outside'.

The basic BSP tree algorithm loops across all the edges of a polygon and finds those for which one vertex is on each side of the partition plane.

The goal of this study is to develop a parallel Binary Space Partitioning tree algorithm and implement it on a shared memory parallel processor architecture. For implementation we use the SEQUENT machine and its parallel programming library. This parallel algorithm results in increased overall speed.

CHAPTER II

REVIEW OF LITERATURE

Hidden Surface Elimination Algorithms

The hidden surface computation determines which objects in the simulated environment are visible, and which are obscured. For this algorithm many realistic image processing algorithms have been developed in university and company environments over the past years. In this section we review several hidden surface elimination algorithms: Ray tracing, Painter's algorithm, Z-buffer algorithm, Subdivision algorithm, and BSP tree algorithm.

(A) Ray Tracing

Ray tracing is one of the methods in the 2-D and 3-D graphics. The earliest ray tracing algorithms were based on the brute force technique. These algorithms solve the ray environment intersection problem, finding the closest point of intersection between an arbitrary ray and the objects in the environment. An attempt is made to intersect the ray with each of the objects in the environment. The resulting intersections are sorted to determine the closest one.

Although ray tracing provides powerful basis for realistic image computation, it traditionally has been associated with high time complexity and unstructured environments [7]. Because image generation involves both hidden surface computation and shading computation, time complexity is very high.

(B) Painter's Algorithm

When we make a realistic image using polygons, the problem of overlapping polygons will occur. The major issue with overlapping polygons is the order in which they are filled. Implicitly the polygons have a priority ordering. Each polygon with higher priority paints over any polygon of lesser priority. This method is called painter's algorithm.

In a painter's algorithm, surfaces are scan-converted in reverse-priority order. In the reverse priority order, the most distant surface is written to the frame buffer first. Subsequent surfaces are written over earlier surfaces to hide them [13].

Unfortunately, when neighboring polygons are rendered, a problem occurs. The previously painted pixels will be given a blended color on the new polygon. Therefore, some of the background painting will show polygon edges.

(C) Z-buffer and Subdivision Algorithm

A number of algorithms have been developed to solve the color blending problem when polygons overlapped. The most popular algorithms are Z-buffer algorithm and Subdivision algorithm.

Z-buffer algorithm represents each display pixel by one element in an array. The array is used to keep track of the distance from the viewpoint to the polygon to be drawn at the point. As a polygon is drawn to the screen, the distance at each pixel is computed and compared with its corresponding array element. Finally a new polygon pixel is rendered to the closer viewpoint instead of the previous polygon drawn at that pixel [13].

Subdivision algorithms analyze a group of polygons, breaking them into smaller pieces that do not overlap. This requires drawing the polygons concurrently and splitting each scan line into nonoverlapping horizontal sections. Alternatively, each polygon may be checked against every other polygon for overlap. Thus both approaches require significant extra processing [14].

(D) Binary Space Partitioning Tree Algorithm

Another algorithm that attempts to speed up object space hidden surface image generation is Fuchs' BSP tree algorithm [1]. This algorithm preprocesses the entire scene, classifying the objects in the environment into a binary space partitioning tree. Traversal of the resultant

data structure produces a list of scene elements in a visually consistent back to front ordering.

BSP tree is a binary tree that represents a recursive partitioning of n-space until space is empty [21]. In three -space, arbitrarily oriented planes partition the scene. The back to front rendering order is determined by an in-order tree traversal. It is handled only by the position of the viewer and no sorting key is need. The BSP tree algorithm deals with the problem of cyclic overlaps and polygon interpenetration by splitting the polygons during the initial construction of the BSP tree.

(E) Comparison of Algorithms

In painter's algorithm, some problems occur in polygon rendering. First, the center of a large polygon may be closer to the view point than a smaller one, yet the large polygon should be drawn before the small one. Second, when three polygons overlap in a circular manner, the painter's algorithm will be incorrect, irrespective of the order in which the polygons are drawn.

While Z-buffer algorithm is a good solution for color blending, it requires a large size memory. Each pixel may require 16 or 32 bits to represent accurately the distance, and the extra distance computation can be time intensive.

Ray tracing algorithms analyze a group of polygons, breaking them into smaller pieces that do not overlap. This

may entail drawing the polygons concurrently, splitting each scanline into nonoverlapping, horizontal sections. Alternatively, each polygon may be checked against every other polygon for overlap. Like Z-buffer algorithm, ray tracing algorithm requires significant extra processing [12][14].

BSP trees were developed to determine the visibility of surfaces. They were later adapted to represent arbitrary 2-D and 3-D shapes. BSP trees are built during a preprocessing step for very fast polygon sorting and making excellent images. BSP tree is a good realistic image sorting algorithm.

Parallel Processing Model

Usually a tree structure is built from the root node. Intuitively, if we initiate more than one processor to build different nodes of the tree in parallel, we can obtain shorter tree building time.

In single-processor machines, a tree is built sequentially one node at a time. Multiprocessor machines allow several processes to run simultaneously. If we use a parallel programming library (e.g. *fork, m_fork)*, we can make several processes at the same time. On the SEQUENT system, a user can execute several parallel processes, whose number equals that of the number of CPUs [23]. Instead of

building a tree one node at a time, we can build a tree by creating subtrees simultaneously.

SEQUENT systems support the following two kinds of parallel programming : multiprogramming and multitasking. Multiprogramming is an operating system feature that allows a computer to execute multiple unrelated programs concurrently. Multitasking is a programming technique that allows a single application to consist of multiple processes executing concurrently. The parallel programs in this thesis are primarily about multitasking, since the DYNIX operating system does multiprogramming for all user programs automatically.

The multitasking programming in SEQUENT system has two programming methods : data partitioning and function partitioning. Data partitioning involves creating multiple, identical processes and assigning a portion of the data to each process. When we make data partitioning programs, we usually use *m_fork* function. Data partitioning is appropriate for applications that perform the same operations repeatedly on large collections of data. For example, data partitioning is appropriate for applications that require loops to perform calculations on arrays or matrices.

Function partitioning involves creating multiple unique processes and having them simultaneously perform different operations on a shared data set. Function partitioning is suitable for applications which must perform many different

operations on the same data. Therefore, function partitioning is appropriate for applications that include many unique subroutines or functions. We can usually use *fork* function for function partitioning programs [23].

This study uses only *m_fork* function for parallel programs, because *m_fork* function can create more than 2 processes that are executed in parallel. Also, it is more efficient than *fork* function [23].

CHAPTER III

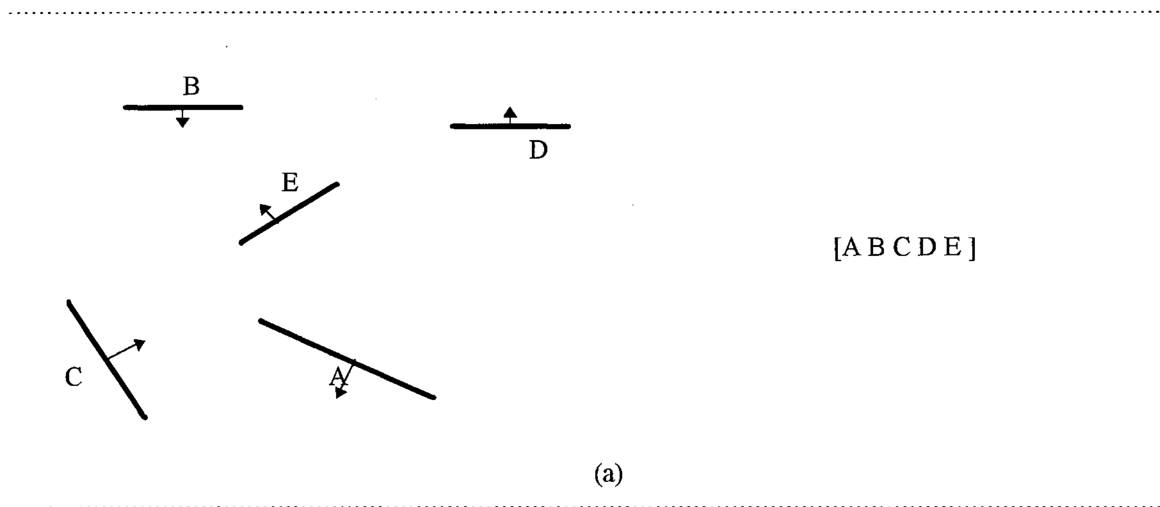
BSP TREE OPERATION

Generating a BSP tree is conceptually straightforward. From a list of polygons, choose one polygon to be the root. Then separate the remaining polygons into two groups : one in front and the other behind the plane of the root polygon. Next, recursively build BSP tree out of each group. The root of each subgroup becomes a child of the root containing it. If any polygon is not completely in front of or behind the plane of the root, it is split into two polygons. The following defines the criteria for choosing a root polygon : The best choice causes the fewest numbers of splits in the remaining polygons.

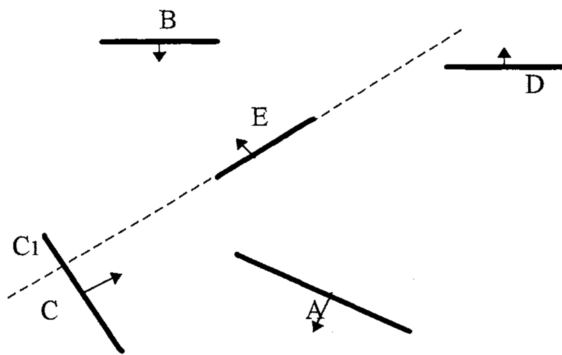
As described above, a BSP tree represents a recursive, hierarchical partitioning of n-dimensional space. BSP tree construction is a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive applications of the method.

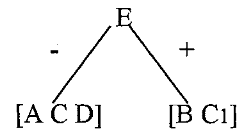
(A) Building BSP tree

The BSP tree is constructed only for a given static scene. First, a polygon is selected. Any polygon can be selected. Its plane partitions the scene into two half-spaces. One half-space contains all remaining polygons in the positive side of this root polygon, relative to its plane equation; the other contains all polygons in its negative side. Polygons that intersect the plane are split by the plane, and their positive and negative pieces are assigned to the appropriate half-spaces. This process recurs within each half-space until that space is empty. An example of a sequence steps in BSP tree construction is shown in Figure 1. Chin's modified algorithm is used for tree construction [21].

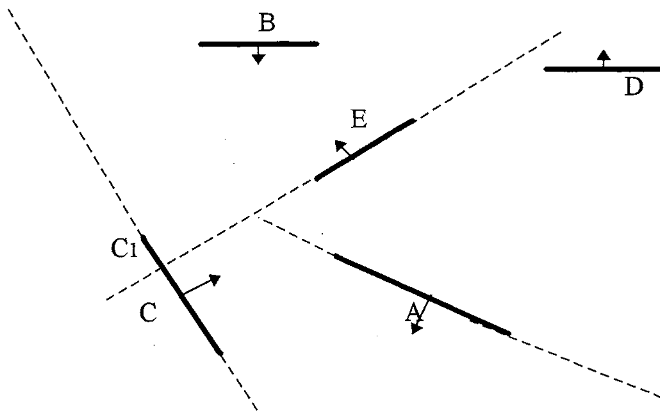


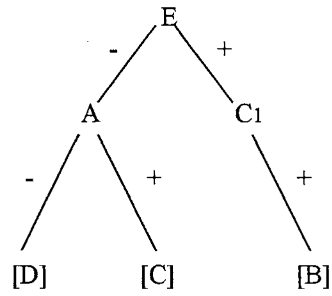
(a)





(b)





(c)

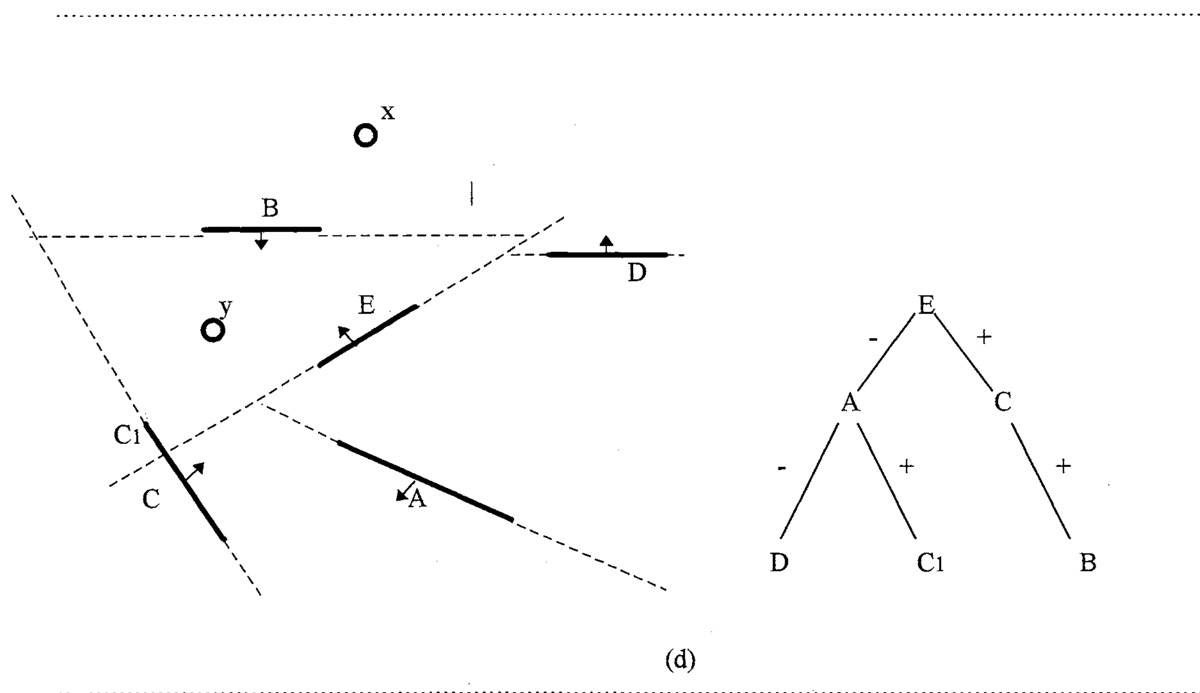


Figure 1. BSP tree construction(adopted from [21]).

The above construction shows both the geometry of the scene and corresponding BSP tree at successive steps. The scene shown in Figure 1(a) with six polygons, is depicted in 2-D as lines. Arrows represent their surface normal with the arrowhead indicating the direction of the positive half-space. The - and + signs represent the respective negative and positive BSP tree branches. The circled letters represent polygons yet to be processed for that half-space, that is, unassigned nodes.

First, select polygon E to define a root partitioning plane. It partitions the scene into two half-spaces as indicated by the thin line in Figure 1(b). One half-space contains all the remaining polygons in its positive side, i.e., B. The other half-space contains all the remaining polygons in its negative side, i.e., A, and D. Since C intersects the partitioning plane, it is split into C and C1. Distribute each portion of C into the appropriate half-space(Figure 1(b)). Node E becomes the root; its two branches each contain a list of polygons yet to be processed for its corresponding half-spaces.

This process is continued recursively by choosing another plane within each half-space to partition the remaining polygons. This continues until no plane remains, as in Figure 1(c) and Figure 1(d).

After building BSP tree, we can make back-to-front polygon order.

(B) Traversing BSP tree

The BSP tree's greatest advantage is that a special in-order traversal of the tree is possible. This traversal provides for an O(n) back-to-front ordering of polygons from an arbitrary viewpoint. This traversal recursively does the following. To render polygon P, first all of the polygons in P's half-space opposite to the viewer are rendered, then P

is rendered, then all of the polygons in P's half-space containing the viewer are rendered.

For an illustrative traversal, assume that the viewpoint is x in Figure 1(d). First, x is on the positive side of node E, so traverse the negative side of node E. Next, x is on the negative side of node A, so A is traversed. Next C is traversed since it is the only polygon there. Next, render node A's polygons. After that, D is rendered because D is in the negative side of node A. Next, traverse the positive side of node E. As C1's negative space is empty, render C1. Finally, traverse the positive side of node C1, rendering B. The complete back-to-front ordering is [C, A, D, E, C1, B].

If view-point is surrounded by polygons, as in the case of viewpoint y in Figure 1(d), the back-to-front ordering is generated as follows. First, y is in the positive side of node E, so traverse its negative side toward node A. Since y is in A's negative side, traverse A's positive side, rendering C. Return to render A. Traverse A's negative side, rendering D. Return to render E. Next, traverse the positive side of E. Since y is in the positive side of C1 and its negative branch is empty, render C1. Traverse C1's positive side, finally rendering B. The ordering is [C, A, D. E, C1, B].

CHAPTER IV

PARALLEL PROGRAMMING MODEL FOR BSP TREE CONSTRUCTION

Use of BSP tree in computer graphics involves three important procedures; read polygon list, build tree, and draw polygon on the monitor. Among these procedures the important part is to build tree procedure. The emphasis of this thesis is on this part. In this chapter, we first give a known sequential algorithm and then provide a new parallel algorithm.

(A) *SEQUENTIAL* Algorithm for Tree construction and display

Following is a sequential algorithm for building a BSP tree [7].

Algorithm : BSP-tree-build

```
BSP_tree *BSP_maketree(polygon *polylist)
{
    polygon  root;
    polygon  *backlist   *frontlist;
    polygon  p,  backpart,  frontpart;
    if (polylist == NULL)
        return  NULL;
    else {
        root =   and remove poly(&polylist);
        backlist = NULL;
        frontlist = NULL;
        for (each remaining polygon p in polylist) {
              if (polygon p in front of root)
```

```
                BSP_add to list (p, &frontlist);
         else if (polygon p in back of root)
                BSP_add to list (p, &backlist);
          else {     /* polygon p must be split */
         BSP_splitpoly (p, root, &frontpart, &backpart)
                BSP_add to list (frontpart,&frontlist);
                BSP_add to list (backpart, &backlist);
            }
      }
      return BSP_combine tree(BSP_maketree(frontlist),root,
                                   BSP_maketree(backlist));
   }
} /* BSP_maketree */
```

After building BSP tree, we can draw polygon images on the screen. For drawing polygons, we need to calculate the visibility priorities. Calculation of polygon order is a variant of an in-order traversal of the environment's BSP tree(traverse one subtree, visit the root, traverse the other subtree). We must have an order of traversal that visits the polygons from those farthest to the closest to the current viewing position. At any given node, there are two possibilities: positive side subtree, node, negative side subtree or negative side subtree, node, positive side subtree. We choose one of these two orderings based on the relationship of the current viewing position to the polygon of node. Specifically, we are interested in the side of the polygon of node where the current viewing position is located.

The traversal for a back-to-front ordering is 1) the negative side, 2) the node and 3) the positive side. This notion of a traversal may be embodied in at least two

different ways for visible surface image generation. The first way is to assign priorities to polygons in the order that we visit them. Using the in-order traversal we will get a low-to-high visibility priority. The second way does not assign explicit visibility priority value to polygons. But this way uses the painter's algorithm that each polygon paints over any polygon of lesser priority. Since higher priority polygons are visited later, they will overwrite any overlapping polygons of low priority. The following recursive procedure generates a visible surface image in the above described manner [1][21]:

Algorithm : BSP-tree-display

```
BSP_displaytree(BSP_tree  *tree)
{
    if (tree != NULL) {
        if (viewer is in front of tree -> root) {
          /* display back child, root, and front child */
             BSP_display tree(tree -> backchild);
             display polygon(tree -> root);
             BSP_display tree(tree -> frontchild);
         } else  {
           /* display front child, root and back child */
            BSP_display(tree -> frontchild);
            displaypolygon(tree -> root);
            BSP_display tree(tree-> backchild);
          }
      }
 } /* BSP_display tree */
```

(B) Parallel Algorithm for tree Construction

To my knowledge, there is no work reported in the open literature about parallel algorithms for building BSP tree in multiprocessor environment. If we want to implement a parallel BSP tree building algorithm on a shared memory system, we do not have to make major change to the tree representation scheme described in the previous section because the complete tree list can be stored in a shared memory. However, new algorithms should be developed to suit the nature of the shared memory system.

As mentioned in chapter II, using parallel programming functions we can make two kinds of parallel programming algorithms : function partitioning algorithm and data partitioning algorithm. The approach taken in this thesis is function partitioning. Function partitioning algorithms as applied to tree construction is described below.

First, a program reads all input data and creates an array of polygons. One of the polygons is chosen as the root node. Split function compares all input data with root data and splits the data into two halves. Then we have two lists for input data. One is the left side of the root(data < root) and the other is the right side of the root(data > root). As Figure 1 shows, one part is located at the front side of the root and the other side is located at the back of the root. Now there are two sets of data to be partitioned. So, create two more processes to partition the two halves. This method is continued until there is no more data to be partitioned. Each process sends the root node to

its parent before termination. When all processes terminate, the tree construction is complete.

The parallelization model used in this thesis combines the idea of function partitioning as described above with the ides of mapping Processors to tasks. Implementation is achieved using *m_fork*. The model is explained below using an example:

Assume that there are four Processors available for computation. The process decomposition of function partitioning is shown in Figure 2.

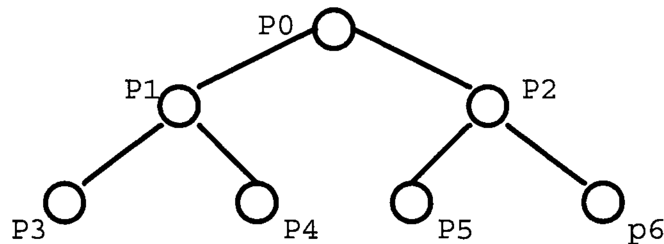


Figure 2.

Process decomposition of build-tree Algorithm.

The process decomposition depends on the number of Processors available for computation. If N Processors are available, we use 2N-1 processes. The processes are organized into a tree. In this example there are seven processes named P0 - P6. P0 partitions the original list.

After each partition, the left side is partitioned by the left child and the right side is partitioned by the right child. When the partitioning reaches the leaf nodes, each leaf node is responsible for construction of the tree based on the list it receives. In figure 2, P3, P4, P5 and P6 represent the leaf nodes. Upon completion, P3, P4, P5 and P6 send the trees to P1 and P2. P1 and P2 construct their trees and send them to P0. P0 completes the construction.

Let P_0, P_1, P_2, and P_3 be the four Processors. Then, the mapping of processes to the Processors is shown in Table 1.

Table 1. Process to Processor mapping.

Processors	**processes**
P_0	P0, P1, P3
P_1	P2, P5
P_2	P4
P_3	P6

In the mapping scheme, Processor P_0 completes the first partitioning. Now, two Processors can be used because there are two lists to be partitioned. So, P_0 takes one list and uses P_1 to partition the other list. When P_0 and P_1 complete the partitioning of their respective lists, two other

Processors can be used. This results in the Processor allocation shown in Table 1. In general, when Processor P_i completes the split, it keeps one half and allocates the other half to Processor $P_{(i+2^h)}$ where h is the level of the node currently being split. The general mapping of Processors to processes is shown in Figure 3. The Processors P_i and $P_{(i+2^h)}$ are called level *h* buddies or level_buddies(buddies for short). Two buddies build the two children of a node at level *h* - 1.

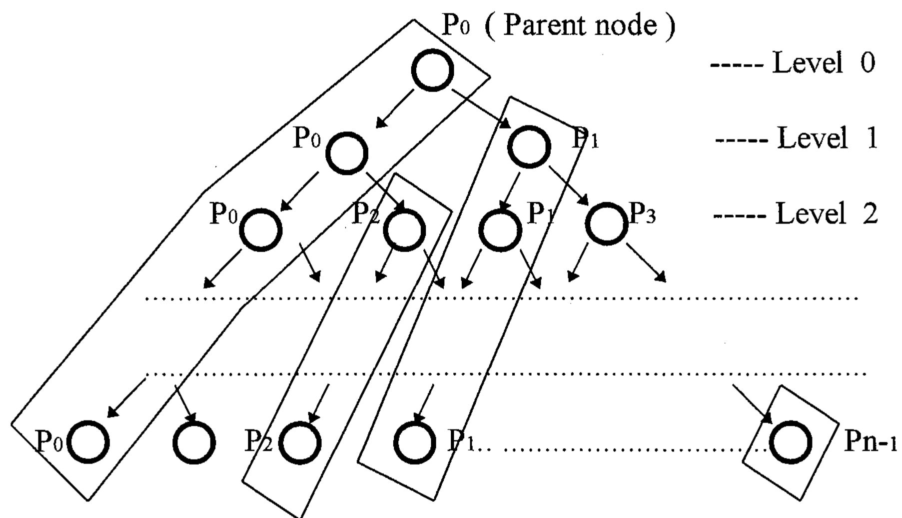


Figure 3. Mapping of n Processors.

The Processor mapping described above is used in our parallel algorithm. Using *m_fork*, the scheme can be described as follows :

Each parallel build-tree process is the same function. For example, if we want to use four Processors to build tree, we use four identical functions because the *m_fork* system call allocates the same function to the four Processors. In general assume that we have N Processors available for computation. Then we use a switch statement where case i represents Processor i. The code associated to case i will be performed only by Processor i. As shown in Figure 3, each processor will be associated to several nodes in the process tree. The actions performed at each node can be abstractly represented by two roles. They are parent and child. The actions performed in each are

Parent's task :

1. subdivide the list.
2. keep one half and give the other half to a child. [These two Processors are called level_buddies], and
3. construct the tree(attach children to root) after building subtrees.

Child's task :

1. become a parent, and
2. when done give the root of the tree to parent.

The following algorithm is based on the scheme

described above :

Algorithm : Parallel BSP-tree-build

```
switch(Processor i){
   case i :
      wait for level_buddy;  /* This Processor waits until
                                spliting reaches its level*/
      while(more-to partition){
          split;
          if(buddy-available)
             signal_buddy;
          else  exit;
          push  root;
          push level;
      }
      if(more_to_partition)
          use  single_Processor scheme;
      while(stack_not_empty){
          pop root;
          pop level;
          receive sibling from level_buddy;
          build_tree;
      }
      sent root to buddy;
      break;
}
```

CHAPTER V

PERFORMANCE ANALYSIS

In this chapter we analyze the performance of the parallel algorithm. First we provide a theoretical analysis. Then we describe an experimental analysis. The two metrics used for analysis in this thesis are speedup and utilization. The experimental performance analysis is based on experiments run with different tree node sizes. Execution time for various operations are measured. To measure the time, we use average elapsed time for 10 random operations in each program.

Figure 4. Polygon shape.

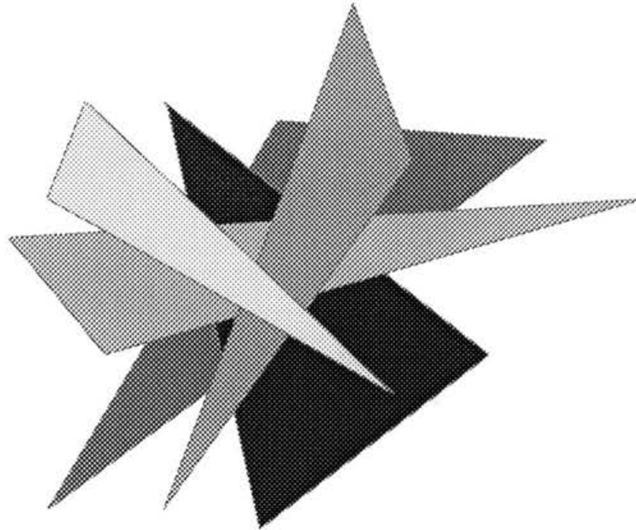


Figure 5. Polygon set after back to front sorting.

I used different sets of triangles for conducting analysis. The triangles are filled triangles. Two examples are shown in Figure 4. To verify correctness of implementation, the polygons were displayed in a back to front order using SPHIGS in an X-terminal. A typical output is shown in Figure 5. The theoretical and experimental analyses of the algorithm are given in the next two sections.

(A) Theoretical Analysis

For the purpose of this analysis, we make the following assumptions :

(1) $N = 2^h$ be the number of processors,

(2) the splitting can be done perfectly and the processes from a balanced binary tree with 2N - 1 nodes, and

(3) T_i be the time for the ith process.

Then,

Time sequential :

$$T_0 + T_1 + T_2 + \ldots + T_{2N-2} \geq (2N - 1) * \min(T_i)$$

In the parallel scheme, the total time is the same as the time spent by Processor P_0.

Time Parallel :

Time spend by Processor P_0 for one level $\leq \max(T_i)$

So, total time spent by Processor $P_0 \leq h \max(T_i)$ where h is the number of levels in the tree.

$$\text{So, Speedup} = \frac{sequential - time}{enhanced - time} \geq \frac{(2N-1)\min(Ti)}{h\max(Ti)} \geq \frac{(2^{h+1}-1)\min(Ti)}{h\max(Ti)}$$

Next we compute the Processor utilization. With the above assumptions, we have the scenario portrayed in table 2.

Table 2. Tree level and Number of Active Processors.

Level	**# of active processor**
0	1
1	2
2	4
3	8
.	.
.	.
h	2^h

Assume that every processor spends the same amount of time at each level. Then at level 0, Processors $P_1 \ldots P_{n-1}$ are idle $\frac{1}{h}$ of the time, at level 1, Processor $P_2 \ldots P_{n-1}$ are idle $\frac{1}{h}$ of the time and so on.

Thus Processor P_0 is utilized 100% of the time, Processor P_1 is utilized $(1 - \frac{1}{h})$ of the time, Processor P_2 and P_3 are utilized $(1 - \frac{2}{h})$ of the time and so on.

Let the number of Processors = N = 2^h where h is the height of the tree.

Utilization = $\frac{1}{N}\left[\sum_{i=0}^{n-1} Time_{busy}(P_i)\right]$

$$=\frac{1}{N}\left[1+\left(1-\frac{1}{h+1}\right)+2\left(1-\frac{2}{h+1}\right)+2^2\left(1-\frac{3}{h+1}\right)+\cdots\cdots+2^{h-1}\left(1-\frac{h}{h+1}\right)\right]$$

$$=\frac{1}{N}\left[1+1+2+2^2+\cdots\cdots+2^{h-1}\right] - \frac{1}{N(h+1)}\left[1+2*2+3*2^2+\cdots+h*2^{h-1}\right]$$

$$=\frac{1}{N}\left(1+\frac{2^h-1}{2-1}\right)-\frac{1}{N(h+1)}\left[(h-1)2^h+1\right]$$

$$=\frac{1}{2^h}\left[2^h\right]-\frac{1}{2^h(h+1)}\left[(h+1)2^h-2^{h+1}+1\right]$$

$$=1 - \left[1-\frac{2^{h+1}}{2^h(h+1)}+\frac{1}{2^h(h+1)}\right]$$

$$=\frac{2}{h+1}-\frac{1}{2^h(h+1)}$$

(B) Experimental Analysis

For each algorithm, we conducted the experiments in four cases, and in each case I created a different number

of processes to execute the algorithm. Also, different input sizes were used. For comparison and analysis, I also recorded the result of executing the single process algorithm. We list the experimental results of the algorithms in Tables 1, 2, 3, and 4. The time used for this analysis is elapsed time in all cases. We use two criteria for evaluation. They are speedup and efficiency. Speedup is the ratio of Ts(the execution time for the sequential algorithm) to Tp(the execution time for the parallel algorithm).

Experiments were conducted with 4, 8, and 16 Processors. The machine used for this research has a limit of 24 Processors. Since 16 is the highest power of 2 less than 24 I used the above mentioned number of Processors.

Efficiency is the ratio of speedup to *N*, the number of Processors executing the algorithm. Designers of parallel algorithms hope to achieve high speedup and efficiency. Maximum possible efficiency is 1. For the sake of clarity, the formulas used for computation of speedup and efficiency are shown below :

$$\text{Speedup} = \frac{Ts}{Tp}$$

$$\text{Efficiency} = \frac{Speedup}{N}$$

Results given in tables 3 - 6 show that speedup increases as the number of Processors increase. Efficiency decreases as the number of Processors increase. This is due to the fact that several Processors wait idle until spliting reaches corresponding level in the BSP tree. Since the shared memory Processors are not massively parallel in general, the limit of 16 Processors used this study does not skew the results.

Table 3. Execution Time, CPU time, and efficiency of 500 polygons

Programming Method	Single Processor	Parallel Processor		
No. of Processes	1	4	8	16
Elapsed Time	2:14.60	1:52.22	1:18.33	54.39
CPU Time	1:58.22	1:18.49	1:03.68	50.42
Speedup		1.20	1.72	2.47
Efficiency		0.29	0.21	0.15

Table 4. Execution Time, CPU time, and efficiency of 1000 polygons				
Programming Method	Single Processor	Parallel Processor		
No. of Processes	1	4	8	16
Elapsed Time	6:52.37	5:01.44	4:12.51	3:03.12
CPU Time	5:52.46	4:07.95	3:40.21	2:12.37
Speedup		1.37	1.63	2.25
Efficiency		0.34	0.20	0.14

Table 5. Execution Time, CPU time, and efficiency of 2000 polygons				
Programming Method	Single Processor	Parallel Processor		
No. of Processes	1	4	8	16
Elapsed Time	10:53.32	8:47.34	7:22.53	5:21.35
CPU Time	9:23.19	7:39.11	6:47.24	4:48.23
Speedup		1.23	1.46	2.03
Efficiency		0.30	0.18	0.13

Table 6. Execution Time, CPU time, and efficiency of 5000 polygons				
Programming Method	Single Processor	Parallel Processor		
No. of Processes	1	4	8	16
Elapsed Time	32:52.37	24:01.45	18:42.31	14:02.46
CPU Time	28:23.19	20:53.33	16:10.37	11:48.21
Speedup		1.36	1.75	2.34
Efficiency		0.34	0.22	0.14

CHAPTER VI

CONCLUSION

In this thesis, we reviewed BSP tree algorithm for polygon generation. The time consuming part of this scheme is the BSP tree construction algorithm. Based on the sequential algorithm found in the literature, we provide a parallel BSP tree building algorithm. In order to develop the parallel algorithm, we developed a mapping scheme of processes organized as a binary tree to Processors. 2N - 1 processes are mapped into N Processors.

The parallel algorithm is implemented in a SEQUENT-S81 shared memory machine. Experiments were conducted to study the performance of the algorithm. SPHIGS is used for displaying 3D polygons. Four lists of polygons were displayed using the algorithm. For each set of polygons, the algorithm has been used with 1, 4, 8, and 16 Processors. The results of the experiments show that speedup increases while utilization decreases. This behavior of the algorithm is due to the implementation scheme used which is busy-waiting. The experimental findings are consistent with the theoretical analysis.

The major draw back of the parallel algorithm is in the area of resource utilization. Future work will be directed towards finding methods to improve Processor utilization. Another problem is to determine the optimum number of Processors using both speedup and utilization as the determining parameters.

REFERENCES

[1] Fuchs, Henry, `On Visible Surface Generation by a Priori Tree Structure', SIGGRAPH, 1980, Conference Proceedings, July, pp. 14-18.

[2] Rubin, Steven M. & Whitted, Turner, `A Three-Dimensional Representation for Fast Rendering of Complex Scenes', ACM SIGGRAPH, 1980, pp. 110-116.

[3] Kunii, Tosiyasu L. & Wyvill, Geoff, `A Simple But Systematic CGS System', Candian Information Processing Society, 1985, May, Proceedings of Graphics Interface, pp. 27-31.

[4] Roth, Scott D, `Ray Casting for Modeling Solids', Computer Graphics and Image Processing 18, 1982, pp. 109-144.

[5] Browne, James C. & Hyder, Syed I., `Visual Programming and Debugging for Parallel Computing', IEEE Parallel & Distributed Technology, 1995, Spring, pp. 75-83.

[6] Parker, J. R. & Ingoldsby, T. R., `Design and Analysis of a Multiprocessor for Image Processing', Journal of Parallel and Distributed Computing 9, 1990, pp. 297-303.

[7] Foley, James D, & Dam, Andries Van, et al., `Computer Graphics: Principles and Practice', Addison-Wesley, 1993, pp. 548-693.

[8] Rogers, David F., & Earnshaw, Rae A., `Techniques for Computer Graphics',Springer-Verlag, 1987, pp. 173-190.

[9] Angel, Edward, `Computer Graphics', Addison-Wesley, 1990, pp. 361-354.

[10] Leendert, Ammeral, `Interactive 3-D Computer Graphics', John Wiley & Sons, 1988, pp. 237-249.

[11] Quinn, Michael J., `Designing Efficient Algorithms for Parallel Computers', McGraw-Hill, 1987. pp. 165-171.

[12] Green, Stuart, `Parallel Processing for Computer Graphics', The MIT Press, 1991. pp. 43-44.

[13] Roregs, David F., & Earnshaw, Rae A., `Techniques for Computer Graphics', 1990. Springer-Verlag, 1987, pp. 249-254.

[14] Naylor, Bruce F., `Partitioning Tree Image

Representation and Generation from 3D Geometric Models', Proceeding of Graphics Interface, 1992, May, pp. 201-212.

[15] Chrysanthou, Y. & Slater, M., `Computing Dynamic Changes to BSP tree', Computer Graphics Forum (Eurographics' 92 Proceedings), 11(3), 1992, Sept, pp. 321-332.

[16] Paterson, Michael S. & Yao, F., `Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling', Discrete and Computational Geometry, 5(5), 1990, pp. 485-503.

[17] Gordon, Dan & Chen, Shuhong, `Front-to-Back Display of BSP Trees', IEEE Computer Graphics and Applications, 11(5), 1991, pp. 79-85.

[18] Sung, K. & Shirley, P., `Ray Tracing with the BSP Tree', Graphics Gems III, AP Professional(Academic Press), 1992, pp. 271-274.

[19] Machover, Mark, `Advanced Computer Technology : The Experts'View' Electronic Digest, 32(3), 1984, pp. 212-213.

[20] James, C. & Syed, I., `Visual Programming and Debugging for Parallel Computing', IEEE Parallel & Distributed Technology, 1995, pp. 75-83.

[21] Paeth, Alan W., `Graphics Gems V' AP Professional (Academic Press), 1995, pp. 121-138.

[22] George, K. M. & Alfantookh, A., `Implementation of 2-4 Finger Trees in The Hypercube Architecture' Proceeding of ACM Symposium on Applied Computing, 1995, pp. 198-205.

[23] Osterhaug, Anita, `Guide to Parallel Programming' Prentice-Hall, 1989, pp. b6-b9.

[24] Wang, Fangju, `A Parallel Intersection Algorithm for Vector Polygon Overlay' IEEE Computer Graphics & Applications, 1993, pp. 74-81.

[25] Whitman, Scott, `Dynamic Load Balancing for Parallel Polygon Rendering' IEEE Computer Graphics & Applications, 1994, pp. 41-48.

[26] Ma, Kwan-Lin, `Parallel Volume Rendering Using Binary-Swap Compositing' IEEE Computer Graphics & Applications, 1994, pp. 59-68.

APPENDIX A

ORGANIZATION OF THE PROGRAM

The BSP tree program I used is composed of seven modules. The following is a synopsis of the modules :

1. BSP allocation :

   The purpose of this module is to allocate, free, and append vertices and faces.

2. BSPcollision :

   This module detects collision between the viewer and static objects in an environment represented as a BSP tree.

3. BSPmemory :

   This module allocates and frees memory.

4. BSPpartition :

   This module partitions a 3-D convex face into two with an arbitrary plane.

5. BSPtree :

   This module constructs and traverses a BSP tree.

6. BSPutility :

   This module computes equation of a plane, normalizes a vector, and performs cross products.

7. main :

APPENDIX B

DERIVATION OF A FORMULA

Proposition :

$$1 + 2 * 2 + 3 * 2^{2} + \cdots + h * 2^{h-1} = (h - 1)2^{h} + 1$$

Proof :

$$X + X^{2} + X^{3} + \cdots + X^{h} = X[1 + X + \cdots + X^{h-1}]$$

$$= X\frac{X^{h}-1}{X-1}$$

$$= \frac{X^{h+1}-X}{X-1}$$

Differentiate both sides with respect to X.

$$1 + 2X + 3X^{2} + \cdots + hX^{h-1} = \frac{(X-1)\left[(h+1)X^{h}-1\right]-\left[X^{H+1}-X\right]}{(X-1)^{2}}$$

$$= \frac{(X-1)\left[hX^{h}+X^{h}-1\right]-X^{h+1}+X}{(X-1)^{2}}$$

$$= \frac{hX^{h+1}-hX^{h}-X^{h}+1}{(X-1)^{2}}$$

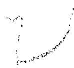
Let $X = 2$,

Then,

$$1 + 2 * 2 + 3 * 2^{2} + \cdots + h2^{h-1} = h2^{h+1} - h2^{h} - 2^{h} + 1$$

$$= h2^{h}[2 - 1] - 2^{h} + 1$$

$$= (h - 1)2^{h} + 1$$



VITA

NAKWON CHU

Candidate for the Degree of

Master of Science

Thesis: A PARALLEL ALGORITHM FOR BINARY SPACE PARTITIONING TREES

Major Field: Computer Science

Biographical:

Personal Data: Born in Seoul, Korea, August 29, 1960, the son of Young-Taek Chu and Sun-Soon Mo.

Education: Graduated from Chung-Ang High School, Seoul, Korea, in January 1979; received Bachelor of Engineering Degree in Mineral and Mining Engineering from Inha University in January 1984; received Master of Science Degree in Mineral and Mining Engineering from Inha University in January 1986; completed requirements for the Master of Science Degree at Oklahoma State University in December, 1996.

Professional Experience: Teaching Assistant, Departments of Mineral and Mining Engineering, Inha University, March, 1984, to December, 1986. Researcher and Analyst, Korea Institute Energy and Resource, Daiduk Research Center, March 1989 to March 1991.