SPATIAL SUBDIVISION FOR PARALLEL RAY CASTING / TRACING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Veysi İşler
February 1995

SPATIAL SUBDIVISION FOR PARALLEL RAY CASTING/TRACING

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

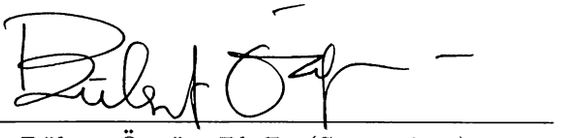
By

Veysi İşler

February 1995

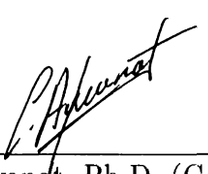
Veysi İŞLER
tarafından Değiştirilmiştir.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



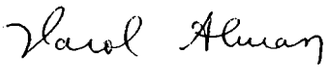
Bülent Özgüç, Ph.D. (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



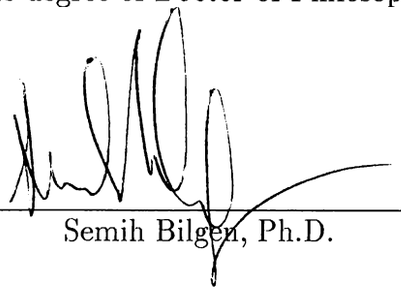
Cevdet Aykanat, Ph.D. (Co-supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



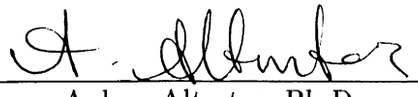
Varol Akman, Ph.D.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



Semih Bilgen, Ph.D.

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.



Ayhan Altıntaş, Ph.D.

Approved for the Institute of Engineering and Science:

Mehmet Baray, Ph. D.
Director of Institute of Engineering and Science

Abstract

SPATIAL SUBDIVISION FOR PARALLEL RAY CASTING/TRACING

Veysi İşler

Ph.D. in Computer Engineering and Information Science

Supervisors:

Prof. Bülent Özgüç and Assoc. Prof. Cevdet Aykanat

February 1995

Ray casting/tracing has been extensively studied for a long time, since it is an elegant way of producing realistic images. However, it is a computationally intensive algorithm. In this study, a taxonomy of parallel ray casting/tracing algorithms is presented and the primary parallel ray casting/tracing systems are discussed and criticized.

This work mainly focuses on the utilization of spatial subdivision technique for ray casting/tracing on a distributed-memory MIMD parallel computer. In this research, the reason for the use of parallel computers is not only the processing power but also the large memory space provided by them.

The spatial subdivision technique has been adapted to parallel ray casting/tracing to decompose a three-dimensional complex scene that may not fit into the local memory of a single processor. The decomposition method achieves an even distribution of scene objects while allowing to exploit graphical coherence. Additionally, the decomposition method produces three-dimensional volumes which are mapped inexpensively to the processors so that the objects within adjacent volumes are stored in the local memories of close processors to decrease interprocessor communication cost. Then, the developed

decomposition and mapping methods have been parallelized efficiently to reduce the preprocessing overhead.

Finally, a splitting plane concept (called "jaggy splitting plane") has been proposed to accomplish full utilization of the memory space of processors. Jaggy splitting plane avoids the shared objects which are the major sources of inefficient utilization of both memory and processing power.

The proposed parallel algorithms have been implemented on the Intel iPSC/2 hypercube multicomputer (distributed-memory MIMD).

Keywords: Ray Casting, Ray Tracing, Spatial Subdivision, Binary Spatial Partitioning (BSP), Splitting Plane, Hypercube Topology, Parallel Processing.

Özet

PARALEL IŞIN DÜŞÜRME/İZLEME İÇİN UZAYSAL BÖLÜMLEME

Veysi İşler
Bilgisayar ve Enformatik Mühendisliği Doktora

Tez Yöneticileri:
Prof. Dr. Bülent Özgüç ve Doç. Dr. Cevdet Aykanat
Şubat 1995

Bu çalışma uzaysal bölme yönteminin paralel bir bilgisayarda gerçeğe uygun görüntü üretmek için kullanılması üzerinde yoğunlaşmaktadır.

Işın izleme çok yararlı olmasına karşın oldukça fazla işlem gerektiren bir yöntemdir. Bu nedenle bir çok araştırıcı, bu yöntemin sorunlarına çözüm bulmak için çalışmaktadır. Bu çalışmalar sonucunda ortaya çıkan parallel ışın izleme yöntemlerinin sınıflandırılması bu tezde yapılmakta, önemli paralel ışın izleme yöntemleri yine bu tezde tartışılmakta ve eleştirilmektedir.

Uzaysal bölümleme yöntemi, bir işlemcinin yerel belleğine sığamayan üç-boyutlu karmaşık sahnelerin ayrıştırılmasına dayanan paralel ışın izleme algoritmasına uygulanmıştır. Geliştirilen ayrıştırma yöntemi, sahnedeki nesnelerin işlemcilere eşit bir şekilde dağıtılmasını sağlamakla birlikte grafiksel tutarlılığın (coherence) kullanılmasına da olanak sağlamaktadır. Uzaysal bölümlemeyi kullanan ayrıştırma yöntemi, ayıran düzlemleri etkin veri yapıları ile oldukça kısa sürede bulmaktadır. Ayrıca, ortaya

çıkan hacimlere bağlı nesnelerin işlemcilere, işlemciler arasındaki iletişimi azaltacak şekilde eşlenmesi de ayrıştırma yöntemi ile eşzamanlı olarak kısa sürede yapılmaktadır. Ayrıca, önişlemde harcanan zamanı azaltmak için, önerilen ayrıştırma ve eşleme işleri de paralelleştirilmiştir.

Son olarak, işlemcilere ait yerel belleklerin tamamını kullanmaya olanak sağlayan yeni bir ayırma düzlemi (çıkıntılı ayırma düzlemi) önerilmektedir. Önerilen çıkıntılı ayırma düzlemi paylaşılan nesnelerin birden fazla işlemcinin yerel belleğinde bulunmasına izin vermeyerek paralel bilgisayarın verimli kullanılmasını sağlar.

Önerilen paralel algoritmalar Intel iPSC/2 hiperküp bilgisayarında gerçekleştirilmiştir.

Anahtar Sözcükler: Işın Düşürme, Işın İzleme, Uzaysal Bölümleme, İkili Uzaysal Bölümleme, Ayırma Düzlemi, Hiperküp Topolojisi, Paralel İşleme.

Acknowledgments

I would like to express my deepest gratitude and thanks to my supervisors Prof. Bülent Özgüç and Assoc. Prof. Cevdet Aykanat for their supervision, encouragement, and invaluable advice in the development of this thesis. I appreciate Assoc. Prof. Cevdet Aykanat for his detailed discussions on the implementation of the parallel algorithms.

I am grateful to Assoc. Prof. Varol Akman for his invaluable comments and suggestions about my research and proposal. I appreciate Assoc. Prof. Semih Bilgen and Assoc. Prof. Ayhan Altıntaş for carefully reading my thesis and offering various suggestions.

I would like to thank to Asst. Prof. Faruk Polat, Asst. Prof. İsmail Hakkı Toroslu, Dr. Uğur Güdükbay, Erkan Tın, Tahsin Kurç and all members of the CEIS Department for their morale support, and to Gülseren Oskay and Bilge Aydın, secretaries of the Engineering Faculty and CEIS Department, for their logistical support.

I would like to extend my deepest gratitude and thanks to my parents and my brother for their morale support. Finally, my sincere thanks are due to my wife and son for their morale support and patience.

The work described in this thesis is partially supported by Turkish Scientific and Technical Research Council (TÜBİTAK) grant EEEAG-5, and Intel Supercomputer Systems Division grant SSD100791-2.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in computing technology have resulted in more complex mathematical models and simulations generating huge data sets. Scientific data visualization techniques should be used in order to discover and exploit the knowledge inherent in such tremendous size of data representing complex phenomena. Scientific data visualization achieves this knowledge transfer or communication operation by both human vision and computer images to provide an efficient method of communication with a very a high bandwidth and an effective interface. The data sets to be visualized come in several forms as the results of simulations, computations or measurements of real models. Molecular models, DNA sequences, brain maps, medical imaging scans, simulations of fluid flow and simulated flights through a terrain are some of the sources of visualized data.

Rendering, which is an important stage within the scientific data visualization pipeline, is the process of producing realistic views of a set of objects in a scene represented by the given data sets. To achieve realism and ease the visual communication of knowledge, the generated views incorporate the interaction of light sources with the objects in the scene to simulate some optical effects such as shadows, reflections, refractions and highlights. Rendering is very crucial, since it consumes too much time due to intensive computations and also determines the quality of the images. Excessive time can be needed for rendering complex scenes represented by a large number of data

sets. Therefore, there has been much research to produce more accurate and higher quality images at faster rates.

The motivations of this research are the excessive time and memory space required for rendering complex scenes. This research tries to solve these problems by exploiting parallelism on massively parallel computers. The rendering techniques used here are the well-known ray tracing and ray casting methods for mathematically defined geometric data. However, the developed algorithms can also be adapted to other rendering methods since they do not exploit any properties particular to ray tracing/casting.

In the initial stages of the research, we have developed parallel ray tracing algorithms based on image-space subdivision where the entire scene is duplicated into the local memories of all processors. The image-space parallel ray tracing can achieve almost linear speed-up since the node processors perform their computations independently and thus do not communicate with each other. However we have seen that linear speed-up may not be achieved easily due to the load imbalance among the processors. Some decomposition and mapping schemes have been investigated. Demand-driven scheme that distributes computations to processors on demand gives the best result compared to others.

Parallel rendering of complex scenes requires the decomposition of both scene data and computations, and mapping them to the processors. This type of ray tracing is called parallel ray tracing based object-space subdivision. The decomposition task has been performed by utilizing the spatial subdivision technique that has been developed for the sequential acceleration of rendering algorithms, particularly ray tracing. The computations and the scene data are mapped to the processors while the decomposition is being carried out. Comparison of this mapping with a graph-based heuristics in terms of interprocessor communication cost has been presented.

Since the decomposition task is computationally expensive for complex scenes, an efficient parallel spatial subdivision algorithm has been developed to decrease the preprocessing time. The mapping task is also performed simultaneously with

decomposition.

The duplication of objects in the local memories of processors is not desirable for efficient utilization of both storage and computation time of parallel computers. For this purpose, we have introduced a new splitting plane, the so called *jaggy splitting planes*, for parallel rendering using spatial subdivision.

In the following two sections (1.1 and 1.2), the rendering methods used, ray tracing and ray casting, and the assumed or target scenes in this research are briefly described. Next, a brief overview of general-purpose parallel architectures and the reasons of choosing MIMD distributed-memory multicomputer are presented.

1.1 Rendering Methods Used

Ray tracing and ray casting are chosen as the rendering methods to produce realistic looking images. Ray tracing and ray casting are image-space computer graphics methods in the sense that each pixel of the image is considered at a time to produce the resultant image. They are briefly described below.

Ray tracing is a popular method for generating realistic images on a computer [34]. This method mainly simulates the interaction of the light sources and the objects in an environment. The light sources are usually assumed to be point light sources. In a naive ray tracing algorithm, a ray, called as primary ray, is shot for each pixel from the view point into the 3-D space. Each object is tested to find the first surface point hit by the primary ray. The color intensity at the intersection point is computed and returned as the value of the corresponding pixel. In order to compute the color intensity at the intersection point, the ray is then reflected from this surface point to determine whether the reflected ray hits a surface point or a light source. If the reflected ray ends at a light source, highlights or bright spots are seen on this surface. If the reflected ray hits another surface of an object, the color intensity at the new intersection point is also taken into account. This gives reflection of a surface on another. When the object is

transparent, the ray is divided into two components and each ray is traced individually. The transmitted ray also contributes to the color intensity at the first intersection point. Shadow comes out at a surface point when no light source in the scene is visible from the surface point. Rays starting from the surface point and passing from each light source are produced and tested if they intersect any objects before reaching the corresponding light sources. The images produced in this way contain reflections, refractions, shadows and shading effects.

Ray casting differs from ray tracing in that the orthographic parallel projection is used instead of perspective projection. That is, the generated primary rays are parallel to each other. Besides, usually no shadow and reflection effects are considered in the produced images. Ray casting is widely used in scientific data visualization that aims to convey as much knowledge in complex scenes as possible to the users. In this research, ray casting is used for very complex scenes which may not fit into the local memory of a processor. On the other hand, ray tracing is used when the scene is not so complex, thus the entire scene data can be stored in the local memory of every processor.

Although the naive ray tracing/casting is a simple algorithm, they require enormous amount of floating point operations. In a naive ray tracing/casting algorithm, the number of objects has a great effect on the whole computation time, since each ray is tested with all objects in the scene to find the first intersection point. The intersection test could be quite expensive depending on the geometry of the object tested. The time consumed for intersection tests may reach up to 95% of the total processing time [34]. Therefore, it is essential to reduce the time taken by intersection tests for producing the images at fast rates.

Most of the research related to ray tracing/casting has been concentrated on the techniques to accelerate the methods for various types of complex scenes containing different objects so that interactive display of more accurate images can be achieved. The research performed for accelerating the methods can be classified into two major categories: sequential and parallel approaches. The accelerating methods are elaborated in Chapter 2.

1.2 Parallel Architectures

Typically, there are two major classes of general purpose parallel architectures on which the rendering algorithms can be developed and implemented: SIMD (Single-Instruction Multiple-Data) and MIMD (Multiple-Instruction Multiple-Data) parallel architectures. In an SIMD architecture, there is a central control unit that assigns a single instruction to all processing elements which operate on different data. MIMD refers to the fact that each processor is executing a set of instructions asynchronously from other processors. MIMD machines are more attractive since different tasks can proceed simultaneously under separate control flow. Since SIMD architectures are particularly effective in problems with data regularities or regular computation requirements, efficient parallelization of our problem can be accomplished on MIMD architectures. Furthermore, MIMD architectures can be classified into two categories in terms of communication type between processors: distributed-memory and shared-memory. Since the performance in shared-memory architectures is limited by the memory contention, most of the current parallel architectures are designed as distributed-memory or hybrid of distributed-memory and shared-memory.

In this research, we have considered the distributed-memory MIMD architecture as the parallel computer since it is well-suited to our problem with irregular data structure and dynamic communication pattern. In a distributed-memory MIMD machine, the speed-up obtained and the memory available can be at most P and $P \times M$, respectively, where P is the number of processors and M is the amount of local memory of each processor. A major advantage of distributed-memory MIMD machines is the large amount of memory space provided that enables very large scenes to be rendered.

An efficient implementation of a parallel ray tracing/casting algorithm requires maintenance of load balance among processors and minimum communication between processors while performing as much work as possible in parallel. There are two main classes of parallel ray tracing/casting algorithms: *image-space* and *object-space* subdivision. In an image-space subdivision, only computations associated with the pixels

of the image are distributed to processors. The scene data is completely replicated in the local memories of processors. In an object-space subdivision, both the scene data and computations are decomposed and mapped to the processors. Each processor stores only a certain portion of the data in its local memory depending on the decomposition and mapping algorithms employed. Although the first one might achieve linear speed-up, the storage of a parallel computer is not used efficiently and thus large scene data cannot be rendered using this scheme. The second scheme allows large scene data to be rendered at the expense of complicating the algorithm. Additionally, the speed-up obtained might not be as high as with the first scheme. To accomplish a high performance in the second scheme that is also called *data parallelism*, the data structures that contain the scene data and support the computations should be decomposed in such a way that each processor is assigned almost equal amount of computations and scene data. The terms data parallelism and object-space based parallelism are used interchangeably throughout the thesis. Here, the key operations are *decomposition* and *mapping*. Collectively, these decomposition and mapping tasks constitute the *domain-mapping* problem. Unfortunately, solution of the domain-mapping problem can be difficult, particularly for large irregular domains [1, 29, 15, 3].

In this research, we have implemented the parallel algorithms on iPSC/2 hypercube which is an MIMD machine. A hypercube of dimension d has 2^d processors labeled as $0..2^d - 1$. Two processors are directly connected if their corresponding binary representations differ in exactly one bit. Hypercube topology can simulate several architectures such as ring, mesh, 3-D array, tree, etc. The iPSC/2 hypercube consists of two main parts: system resource manager and the cube. System resource manager serves as host connected directly to the cube via a high speed channel. It performs program compilation, loading the cube and I/O operations with the cube. The cube contains processors connected together according to the hypercube topology. Each node is composed of an Intel 80386 microprocessor supported by an Intel 80387 floating point co-processor and 4 Mbytes of local memory.

Chapter 2

Acceleration Techniques

This chapter discusses the acceleration techniques developed for parallel ray tracing/casting algorithms. The acceleration techniques can be classified into two according to whether they use uniprocessor (sequential) or multiprocessor (parallel) computers.

2.1 Sequential Acceleration Techniques

Initial approach to speed up ray tracing has been to investigate accelerating techniques on sequential computers. *Bounding volumes* and *spatial subdivision* are two well-known ray tracing acceleration techniques that were also used to develop some other efficient computer graphics algorithms such as hidden surface removal and polygon rendering.

2.1.1 Bounding Volumes

Some simple mathematically defined objects such as rectangular boxes and spheres can be tested for intersection inexpensively in terms of computer time. The complex objects with which the intersection test is costly are surrounded by these simple objects called bounding volumes, and intersections are first tested with the bounding volumes instead

of the complex objects. When the ray intersects the bounding volume of an object, intersection test is done for the complex object as well. Otherwise, intersection test with complex object is avoided. Obviously, the advantage of using bounding volumes is to eliminate the intersection test with a complex object once its bounding volume is found not to intersect with the ray. Its disadvantage is the extra time spent in testing the bounding box if the object itself has a possible intersection. It should be noted that the bounding volumes are not mutually exclusive and thus a ray might be tested for an intersection with more than one object. This is another drawback of the bounding volumes, since an intersection test for a complex object may take excessive time.

When there is a large number of objects in the scene, even the tests for the bounding volumes can take an excessive amount of time. By forming a hierarchy of bounding volumes, a number of tests can be avoided once a bounding volume that surrounds some other bounding volumes is not hit by the ray. Several neighboring objects form one level of the hierarchy. A drawback of this method is that these hierarchies are difficult to generate automatically and manually generated ones can be poor. For instance, a bounding volume that does not surround an associated complex object tightly is poor since more rays tested for intersections will hit the bounding volume but not the complex object. This will result in extra intersection tests with the bounding volumes.

2.1.2 Spatial Subdivision

The other technique to improve the speed of the ray tracing is called spatial subdivision [9, 16]. The 3-D space that contains the objects is subdivided into disjoint rectangular boxes called voxels so that each voxel contains a small number of objects. A ray travels through the 3-D space by means of these voxels. A ray that enters a voxel on its way is tested for intersection with only those objects in the voxel. If there are more than one intersecting object, the nearest point is found and returned. If no object is hit, the ray moves to the next voxel to find the nearest intersection there. This is repeated until an intersection point is found or the ray leaves the largest box that contains all of the

objects. It is necessary, in this case, to build an auxiliary data structure to store the disjoint volumes with the objects attached to them [30, 31].

This preprocessing will require a considerable amount of time and memory as a price for the speed-up in the algorithm. It is, however, worth using the space subdivision particularly when the scene contains many objects, since this data structure is constructed only once at the beginning and is used during the ray tracing algorithm. The number of rays traced depends both on the resolution of the generated image and the number of objects in the scene. The auxiliary data structure helps to minimize the time complexity of the algorithm by considering only those objects on the ray's way.

There are several spatial subdivision techniques that utilize space coherence. They basically differ in the auxiliary data structures used in the subdivision process, and the manner used to pass from one volume to another. There are three major spatial subdivision schemes: octree, BSP (kd-tree), and regular subdivision.

An octree is a hierarchical data structure used for efficiently indexing data associated with points in 3-D space. In the spatial subdivision ray tracing algorithm, each node of the octree corresponds to a region of the 3-D space [10, 11]. The octree building starts by finding a box that includes all of the objects in the scene. A given box is subdivided into eight equally sized boxes according to a subdivision criterion. These boxes are disjoint and do not overlap as the bounding volumes might do. Each of the generated boxes are examined to find which objects of the parent node are included by each child node. The child nodes are subdivided if the subdivision criterion is satisfied. This is carried out recursively for each generated box. The subdivision criteria may be based on the number of objects in the box, the size of the box, the density ratio of total volume that is enclosed by all objects in the scene to the volume of the box.

BSP (Binary Space Partitioning) is a data structure used to decompose the 3-D space into rectangular regions dynamically [16]. BSP is very similar to octree structure in that it also divides the space adaptively. The information is stored as a binary tree (a tree where each non-terminal node can have exactly two child nodes) whose non-leaf nodes are

called slicing nodes, and whose leaf nodes are called box nodes and termination nodes. Each slicing node contains the identification of a slicing plane, which divides all of space into two infinite subspaces. The slicing (splitting) planes are always aligned with two of the Cartesian coordinate axes of the space that contains the objects. The child nodes of a slicing node can be other slicing nodes, termination nodes or box nodes. A termination node denotes a subspace which is out of the 3-D space that does not contain any objects. A box node, on the other hand, is described by the slicing nodes that are traversed to reach it. They denote a subspace containing at least one object. BSP actually encodes the octree in the form of a binary space partitioning tree. The tree is traversed to find the node containing a given point.

Regular subdivision is the last major spatial subdivision scheme for ray tracing. It is simply based on the decomposition of the 3-D space into equally sized cubes [9]. The size of the cubes determines the number of objects in each cube. Therefore, an optimal cube size must be considered such that the overhead for moving through the boxes should not exceed the time gained in testing intersections. One advantage of regular subdivision is the inexpensive traversal of ray through 3-D space to find an intersection point.

2.2 Parallel Acceleration Techniques

A large number of parallel systems have been proposed to exploit the inherent parallelism in the algorithm. Most of these are special-purpose systems that require the construction of custom hardware using VLSI. The recent developments in the VLSI technology have made it feasible to design and implement special-purpose hardware for the ray tracing algorithm [8, 14, 27]. In spite of the gain obtained in this way, these special purpose architectures have several disadvantages. First, there are on-going studies to improve the algorithm itself. Researchers should thus work on general purpose machines in order not to be restricted by the hardware. Second, special purpose hardware is expensive and often restricts the applications that require other computer graphics algorithms.

The other approach that exploits speed-up through the inherent parallelism in ray-tracing investigates the algorithm on a general purpose parallel architecture independent of the hardware configuration [6, 12, 18, 23, 25]. The effective parallelization of the ray tracing algorithm on a multicomputer requires the partitioning and mapping of the ray tracing computations and the object space data. This partitioning and mapping should be performed in a manner that results in low interprocessor communication overhead and low processor idle time. Processor idle time can be minimized by achieving a fair load balance among the processors of the multicomputer. Two basic schemes exist for parallelization. In the first scheme, only ray tracing computations are partitioned among the processors. In the other scheme, both ray tracing computations and object space data are partitioned among the processors.

2.2.1 Image-Space Subdivision

In the first scheme, the overall pixel domain of the image space to be generated is decomposed into subdomains. Then, each pixel subdomain is assigned to and computed by a different processor of the multicomputer. However, each processor should keep a copy of the entire information about the objects in the scene in order to trace the rays associated with the pixels assigned to itself. Hence, an identical copy of the data structure representing the overall object space is duplicated in the local memory of each processor. This scheme requires no interprocessor communication since the computations associated with each pixel are mutually independent.

Assignment of pixels to processors can be either static or dynamic. In the static scheme, the pixel subdomains are assigned to the processors before the execution of the algorithm. However, an even decomposition and assignment of the overall pixel domain do not guarantee an even workload for the processors. The amount of computation associated with an individual pixel may be quite different depending on the location of the pixels and the configuration of the objects in the scene. Furthermore, computational complexity associated with a pixel cannot be predetermined.

1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4
3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8
7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8
7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8
7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8

Figure 2.1: Tiled assignment.

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
5	6	7	8	5	6	7	8	5	6	7	8	5	6	7	8

Figure 2.2: Scattered assignment.

The simplest way of static assignment is *tiled* decomposition where the image space is partitioned evenly into contiguous blocks of pixels and each pixel block is then assigned to a processor. Figure 2.1 illustrates tiled assignment for a 16×16 image space and 8 processors. Most probably, each block will require different amount of computation, which is the source of load imbalance among processors. For example, rays generated at some processors might leave the scene very soon without intersecting any object. These processors will complete their jobs earlier than others resulting in poor processor utilization. Load imbalance problem is solved to an extent by applying *scattered* subdivision which is based on the assumption that adjacent pixels require almost the same amount of computation. Scattered decomposition scheme is achieved by imposing a periodic processor mesh template over the image pixels starting from the top left corner and proceeding left to right and, top to bottom. Figure 2.2 illustrates the scattered decomposition for 16×16 image space and 8 processors. In this scheme, adjacent pixels are assigned to different processors. Hence, this scheme achieves better load balance that distributes the workload to processors more evenly. In this scheme, each processor is responsible for pixels that are *scattered* across the entire image. In the worst case, *scattered* decomposition will behave as *tiled* decomposition that has led to load imbalance. However, such cases are extremely unlikely to be encountered. Therefore, *scattered* decomposition usually performs better than *tiled* decomposition.

In the dynamic scheme, *tiled* decomposition is applied in partitioning the image space assuming very large number of processors. The contiguous pixel blocks are then dynamically assigned to processors on demand. The pool of pixel blocks is resident in a special processor called the *scheduler*. The scheduler is responsible for the assignment of pixel blocks to the demanding processors. The size of pixel block is the number of pixels assigned to a processor on a single request. Each such request demands an extra communication between the requesting processor and the scheduler. Hence, the pixel block size determines the granularity of the distributed computations on the multicomputer. Large pixel block sizes increase the performance of the algorithm by decreasing the number of the communications between the scheduler and the processors. On the other hand, large pixel block size degrades the performance by introducing

load imbalance between processors. For an appropriate granularity, the performance is excellent in terms of load imbalance, since processors are assigned to the computations of a new pixel block as soon as they become idle. This scheme approaches the static *tiled* decomposition scheme as the number of pixel blocks are reduced to the number of processors. The overhead imposed by this scheme is the communication between the scheduler and the processors.

The image-space subdivision achieves almost linear speed-up. No communication is needed between processors. The only overhead is the communication between the scheduler and the processors of the multicomputer. On the other hand, each processor should have access to the whole scene description, since ray-object intersection tests may be carried out with any object in the scene. This is a big disadvantage. Furthermore, sometimes a large amount of storage is needed to hold the object definitions and other related information. Therefore, processors cannot store the entire information about the objects in the scene.

2.2.2 Object-Space Subdivision

In this scheme, the object space data is subdivided and stored in the local memory of node processors. The subdivision of the object space necessitates interprocessor communication, because each processor owns only a portion of the database. During the execution, a processor may need some portion of the database that exists in the local memory of another processor. In this case, either the needed portion is sent to the requesting processor or the ray with the other relevant information is passed to the processor that has the needed part of the database. Thus, we can classify the existing object space algorithms into two: those that are based on the movement of objects between processors [14, 4, 13], and those that are based on the movement of rays between processors [8, 5, 23, 6].

Movement of Model Database

In the first class, the read-only database is distributed to local memories of different processors. Each processor generates a set of rays associated with the pixels assigned to itself. When a processor needs a part of the scene description for intersection tests that is not available in its local memory, a request is sent to the processor that contains this part of the database. The related information is copied or moved to the requesting processor's memory. The local memories behave as a *cache* and contain the object descriptions according to LRU (least recently used) replacement policy. This class of algorithms suffer from communication volume overhead that results from migration of objects between the processors.

Movement of Rays

Two approaches exist in this class. In the first approach, the 3-D space containing the objects is subdivided into several disjoint volumes. The computation related to the objects in a volume is carried out by a specific processor. The ray that travels through 3-D space to find an intersection passes from one processor to another via messages. Each processor contains information about the volume assigned to itself. The intensity calculations for a pixel are performed incrementally by several processors that the ray visits.

The other approach constructs a hierarchy of bounding volumes. The objects in the same bounding volume are stored in one processor. A processor shoots a primary ray and follows it through the hierarchy down to the leaf nodes of this hierarchy that are pointers to the processor in which the appropriate part of the database is stored. If this traversal ends at a pointer to itself, the necessary calculations are performed for the pixel associated with the ray; otherwise, the ray is sent to the concerned processor. Each processor thus controls a block of pixels, the hierarchy, and a portion of the database.

In object-space type of algorithms, the load imbalance is the major problem, since

some processors may contain objects that are more likely to be intersected than others. Additionally, it is not easy to achieve linear speed-up as in image space subdivision where object space data is duplicated in each processor's memory. The communication overhead between processors might drastically effect the performance in the negative direction. This may even result in the deadlock of the system due to a large number of messages traveling around.

Chapter 3

Previous Work on Parallel Ray Tracing

This chapter examines six important papers on parallel ray tracing. Each section below is dedicated to one paper and is organized as follows: First, the key points of the paper under consideration are presented. Next, the paper is criticized according to the proposed algorithm's performance. Finally, some proposals (if any) to improve the system are given. The title of each section is the title of the paper under review.

3.1 Parallel Processing of an Object Space for Image Synthesis Using Ray Tracing

Kobayashi and et al. have designed an architecture for parallel processing of ray tracing [18]. In their design, two types of processors exist for handling intersection calculations and for global shading computations. The first type of processors, called IPs (Intersection Processors), are responsible for intersection calculations and are connected to each other with a hypercube interconnection network. Each processor is allocated a subspace of the 3-D space, and the objects in a subspace are stored in the node processor to which the

subspace is assigned. The second type of processors, called Shading Processors (SPs), are not linked to each other, since they do not need communication. What they do is to calculate the global intensity of each pixel simultaneously. The ray tracing algorithm starts by allocating subspaces and a block of pixels to the IPs. Each processor generates a ray for a pixel and sends it to the relevant processor in which intersection tests are carried out. The subspaces are allocated to the processors so that "face-neighboring" subspaces are in neighboring processors. A ray stops traveling when an intersection with an object is found. At this point, the IP that contains the intersected objects sends the necessary information to the SP which performs the shading using that information. Two other rays might be generated by the IP in the refraction and reflection directions. The same process is applied to these rays. The intermediate intersection results with shading information are sent to the SPs when intersected objects are found. Meanwhile, SPs update the color value of the pixel as soon as they receive a message containing an intersecting ray and the relevant shading parameters.

The remarkable improvement is in the data structure Kobayashi and et al. used to efficiently pass from one subspace to another. They proposed an adaptive division of an object space, and building what they call an adaptive division graph that contains spatial information to pass from one subspace to another. The algorithm to build the adaptive division graph takes an octree as an input and generates the graph in which vertices denote the subspaces and edges between vertices denote the face-neighboring relation.

Although the address of the next subspace is found out by only one reference of a pointer, the graph is about 1.8 times larger than an octree. Their method requires a processor contain the face-neighboring subspaces before the ray tracing starts. When a ray is to be moved to the next subspace, the processors find the address of the processor from the graph by only one reference. Since the processors do not know the smallest size of the 3-D space, several iterations are required to locate the next subspace (right, left, down, up). It may be suggested that each node stores the size of the smallest voxel (subspace) and the next subspace location is found by incrementing this much;

this guarantees no other subspace on the ray's path is skipped.

One disadvantage of their approach is the load imbalance among processors. Unfortunately, no measure is taken for this major problem. It seems that the processors (both IPs and SPs) are not efficiently utilized, because some of them will be idle most of the time, if their objects are obscured by other objects.

Unfortunately, only the time consumed at intersection tests in different schemes are compared, namely naive ray tracing, octree algorithm and adaptive division graph. Therefore, we suspect that the data structure proposed is not suitable for load imbalance problem at all. The proposed data structure cannot perform better than octree data structure when the number of objects is not large.

3.2 Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision

As criticized above, Kobayashi and et al. did not treat the load balance problem in their paper. In this paper, the same authors concentrate on the load imbalance problem in a multiprocessor ray tracing system [19].

They try both dynamic and static allocation of 3-D space objects in order to maintain load balance among processors. In both schemes, the 3-D space is subdivided into subvolumes regularly. A 3-DDDA (3 dimensional digital differential analyzer) is used to move a ray in an object space and determine the next subspace to be checked. A 3-DDDA is the extension of DDA which is used to draw a line on a raster grid. Since 3-DDDA finds the next box by means of incremental calculations, movement to the next box is very fast. Their algorithm is based on the movement of rays between processors. In the first static scheme, a block of neighboring subspaces are assigned to a processor. Rays travel in 3-D space via messages. The global shading computation is performed while

intersections are found out. Kobayashi and et al. simulate the proposed distribution of subspaces on a one-dimensional, two-dimensional and 3-D array processors where the first and the last nodes are connected to each other (in a wraparound fashion).

Their second static scheme is an assignment of subspaces which are scattered (distributed) over the entire 3-D space. That is, the subspaces assigned to a processor are not, in general, neighbors of each other. In the tiled assignment, in which one large region is assigned to a processor, the utilization of processors is very low due to the load imbalance among processors. Since the computation in a scene is usually concentrated to some regions of the 3-D space, some processors will have no or few objects to process. Therefore, processors should be responsible for 3-D regions scattered over the 3-D space.

In the simulation, they used processor arrays of dimension 1, 2 and 3. As expected, a 3-D processor array fails in maintaining load balance. This is due to the nature of ray tracing which sends rays from a viewpoint and most probably the objects at the back of the scene will be involved in less intersection and local shading calculations. This means that processors which are responsible for those types of subspaces will become idle most of the time; this leads to the poor utilization of the multiprocessor system.

It is pointed out that when the number of processors increases, the utilization decreases, since the scattered subdivision approaches tiled subdivision. It is true that spatial coherence is not utilized any more, because the processors will probably take care of subspaces far away from each other which have different computational load.

We do not agree with the claim that scattered subdivision performs almost excellent processor utilization when the number of processors is not large. In scattered assignment, they may keep all processors busy by assigning close subspaces to each processor. However, they definitely increase the communications of rays between the processors. Because, a ray will very likely be sent to another processor if no intersection is found in the current processor. The reflected and refracted rays are also very likely to move to another processor. The probability of frequent traveling of rays is high due to the assignment of neighboring subspaces to different processors. Therefore, scattered

assignment is not a good idea in an object-based subdivision in ray tracing. A disadvantage of scattered assignment is the poor utilization of memory space. Since some objects will be duplicated in several nodes, the total memory occupied will be larger than the actual storage for object descriptions and other relevant scene parameters. The total memory requirement in tiled assignment is less than that of scattered assignment, since the neighboring subspaces are allocated to a processor. That is, duplication of objects will be less in total.

In the last section of the paper, it is stated that the effective utilization decreases as the number of processors increases and that it is difficult to utilize the system efficiently. They thus proposed a hierarchical multiprocessor system with static and dynamic load balancing mechanism. The system consists of two levels : a cluster level and a processing element level. At the cluster level, the subspaces are assigned to each cluster by using scattered assignment. That is, rays travel between clusters to find an intersection. At the processing element level, the load assigned to a cluster is carried out in parallel by the processing elements. Stated another way, the clusters are assigned load before the execution whereas the processing elements in a cluster are assigned load in the execution time (dynamically). The simulation results of the proposed architecture seems excellent in terms of both efficiency and speed-up. Almost linear speed-up and an efficiency of 0.9 for several scenes that contain different number of objects are achieved. This is really an excellent result for parallelizing ray tracing. However, the proposed architecture is a special hardware and as discussed before, special purpose architectures are both expensive and restrict other computer graphics applications. Next, the simulation is applied for 4×4 array processors which gives very good results in the static scheme as well. Only the number of processing elements in a cluster is changed in simulation and the number of clusters is kept constant which may be a reason for good timings. It is obvious that when the number of processing elements is increased, the problem of accessing the same object descriptions simultaneously will again be difficult to solve.

3.3 A Self-Balanced Parallel Processing for Computer Vision and Display

Caspary and Scherson use a tree of extents to store the scene description [5]. This tree is then cut at some level and the lower level tree with object descriptions are distributed to the processors. The upper level of the tree containing bounding volumes is duplicated in each processor.

Each processor runs two processes, one data-driven and the other demand-driven. A process is data-driven whenever a task is requested by a processor, the requested processor has to perform the computations using the database it owns. Since the lower tree is distributed to processors, the computation related to this part of the tree should be performed by specific processors. Demand-driven process means that processor request a task to perform on demand whenever its workload is light.

The architecture to implement this algorithm consists of a number of processors connected by a hypercube interconnection network and a host processor that constructs the auxiliary data structure and controls the workload distribution.

The algorithm has three stages. In the first stage, the host processor builds the hierarchy of bounding volumes. Next, in the second stage, the hierarchy is cut from a level and the lower part of the hierarchy is divided into subtrees and the subtrees are distributed to the processors. The upper part of the hierarchy is sent to all processors. The third stage involves the ray tracing algorithm. In this stage, host contains rays to be traced and processors make requests for them. Initially, each processor is assigned a block of pixels for which rays will be generated. A processor traces a ray by first traversing the upper tree which exists in all processors. When traversal ends up at a subtree that is available in the processor, it continues to test for intersections with the ray and the objects in the scene. Otherwise, if the subtree is in another processor, the originator processor makes a (data-driven) request for completing the traversal operation from the processor that has the subtree. This request has higher priority than a demand-driven

request, because no other processor owns the information about the subtree. After the intersection point is found, the originator processor receives the relevant parameters of intersected surface. It makes a request (demand-driven) for extra job from the host.

The key point that gives rise to the load balance is the division of all tasks into two kinds, one of which (demand-driven) can be executed by any processor. The determination of the level where the hierarchy is cut effects the load balance and the utilization of the system. If the level is selected as the bottom of the tree, all processors will have the whole hierarchy which results in the inefficient utilization of memory. If the level is selected near to the root of the hierarchy, the load balance will be difficult to maintain, since data driven task will last longer than the demand-driven task. Most of the bounding volume intersection tests will be carried out by processors that owns the object descriptions. Another consequence of choosing the level low is the increased number of communications between processors.

The algorithm solves the load imbalance problem. The idea of using two types of processes is excellent and leads to almost linear speed-up for moderate scene descriptions. Unfortunately, the algorithm might lead to network congestion due to a huge amount of messages for complex scenes. We may propose to distribute the database considering spatial coherence. In their algorithm, the database is distributed to the processors randomly by the host processor. Instead of this, the host might distribute the adjacent objects to the neighboring processors and the rays should be allocated to the requesting processors according to this distribution. That is, the host may keep several queues that contain different classes of rays.

The next improvement can be the gathering of intermediate shading results in the host processor. In the present algorithm, all intermediate results are accumulated in the processor that originates the ray. For this purpose, a stack is used to store the information about a ray that has reflected or transmitted.

Finally, the construction of hierarchy is difficult and time consuming. This scheme can be applied to the octree.

3.4 Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube

In this paper, Priol and Bouatouch survey some parallel ray tracing algorithms implemented on distributed memory parallel computers [26]. Additionally, a parallel ray tracing algorithm implemented on iPSC/1 hypercube parallel computer is presented. The algorithm is based on the distribution of the database among processors. The load is allocated to processors statically. In their algorithm, effort has been made to avoid deadlock and terminate the distributed algorithm.

The algorithm first subdivides the 3-D space into subvolumes by sub-sampling the image space. The purpose of sub-sampling is to represent a set of coherent rays by only one ray created for a region of pixels. In other words, the image space is partitioned into subimages of size, for example, 8×8 and for each region a primary ray is shot into 3-D space.

The generated sample rays are traced in the 3-D scene and their position and orientation give a criterion to subdivide the 3-D space. The disjoint 3-D volumes are then assigned to processors by mapping an adjacency graph on a hypercube topology. Each processor is also assigned a block of pixels that results from the intersection of the volume with the screen plane. The allocation of volumes to processors takes the 3-D regions as vertices of a graph. The edges between vertices are defined according to the adjacency of the corresponding 3-D regions.

Priol and Bouatouch tried two types of algorithms, namely greedy and iterative, to map the adjacency graph on the hypercube topology efficiently. The vertices of adjacency graph are thought as the processes and the edges are thought as communication between processors. A vertex can be considered as a process, since a process is responsible from the volume assigned to itself. The objective is to minimize the communication between processors.

The global shading is performed by the host processor in order to avoid deadlock of

the system due to a large number of packets traveling between processors.

The termination algorithm they used is the one proposed by Dijkstra [7]. The processors form a ring on which a token moves. The token is initially created by node 0 and sent to its neighbor when all primary rays are generated at this node. The token may be white or black, and initially it is white. The termination is effective when the token remains white after the token completes a tour around the ring.

The results presented are not very good in terms of speed-up and efficiency. As the number of processors increase, the efficiency of the algorithm decreases drastically. Another drawback of the algorithm is that many objects are duplicated in the processors because of the subdivision method used.

3.5 A Parallel Algorithm and Tree-Based Computer Architecture for Ray-Traced Computer Graphics

Green has designed and implemented a tree-based parallel architecture for ray tracing algorithms that distributes the database and tries to maintain load balance among processors [14].

Root processor of the tree stores the entire scene description in its local memory. Initially, the node processors are assigned a block of pixels. When a node fails to find a needed object description in its local memory, it sends a message to its parent requesting the needed object. In this configuration, objects moves from one processor to the other (from parent to child) unlike the scheme where ray travels in the network of processors. Each processor thus uses its local memory as a cache to reduce the communications with other processors (ancestors, parents, children). The replacement policy for object description is based on a LRU (Least Recently Used) algorithm.

In their algorithm, the task allocation as well as database distribution is dynamic. A

node processor requests work from its parent if is finished with all allocated rays. This is achieved by keeping a stack of rays to be traced. The child processor that request a work is assigned one of these rays. Since the objects needed for this ray are already stored in the parent node, communication overhead is not significant. New rays that are generated as a result of reflection are pushed into that stack. An empty stack means that the processor has nothing to do.

The algorithm also gains speed-up by dividing the 3-D space in octree fashion. The octree data structure is duplicated in all nodes to utilize the spatial coherence. They implemented the system with 8 transputers and the speed-up is 4.46 for a scene description containing 2000 spheres. The algorithm was written in Occam language[20].

The first argument is that the system performance would degrade if a tree of more processors, for example 64, were used. In this case, the total communication overhead would increase drastically. The second argument is that the memory is wasted considerably. Since the entire octree data structure and some objects in the scene are duplicated in the node processors. The reason why their system is not very fast may be due to the use of Occam parallel programming language which is a high level language where all communication is synchronized. That is, the processors are interrupted frequently.

3.6 Distributed Object Database Ray Tracing on the Intel iPSC/2 Hypercube

Carter and Teague have developed a parallel ray tracing algorithm that distributes the object space among processors [4]. The algorithm is mainly based on the movement of objects between processors. Some portion of the local database is used as a cache and moved objects stay there until they are replaced by other objects. The algorithm starts by constructing a hierarchy of bounding volumes as Goldsmith did [11]. The upper part of the hierarchy which consists of bounding volumes is duplicated in all processors. The

leaf nodes of the hierarchy that contain pointers to a group of objects are allocated to the processors. At the end of distribution of object database, each node will consume approximately same amount of memory to store the objects. The objects are kept at the local memory of the processors which are allocated at the initialization time until the end of execution. The processors are then assigned a group of pixels which are close to each other. Next, rays are generated at processors and the hierarchy is traversed to find out the first intersection point. If the traversal ends up with a bounding volume the objects of which exist in another processor, a message is sent to the relevant processor to copy the objects to the requesting processor's local memory.

Some part of memory is used for objects that move from other processors. If no memory space is left for new coming objects, the objects resident in these memory cells are replaced by using LRU replacement policy. The objective of moving all objects in a bounding volume is to avoid setup time for short messages and to prefetch the neighboring objects which are likely to be needed in the near future. This is analogous to the notion of line size in a conventional cache memory.

To satisfy the load balance condition, the pixels are assigned to the processors on demand. One point to reduce the communications in the system is to give adjacent pixels to the same processor in order not to need objects which are not present in the local memory of the processor. Their algorithm has changed considerably the ray tracing loop to make the processors busy all the time. This is achieved by ray tracing other rays when the processor is stuck because of waiting for objects in other processors. This makes the ray tracing algorithm highly complex. Each node has a number of states which are changed when some events occur. The transitions are based on the object movement between processors. They solved the control problem by designing a finite state automaton (FSA). The interruptible ray tracing loop is a clever approach to utilize processors efficiently. Unfortunately, message traffic is again high due to movement of objects between processors. The second disadvantage of the method is the difficulty (sometimes impossibility) of the decomposition of the image space so that each processor will have the region of pixels that need almost the same objects. We do not know the

performance of the system for different number of objects and processors. What they gave is the performance of the cache. It would be interesting to move rays which take less time to communicate in addition to the objects in the system. Finally, other replacement policies could be investigated to reduce the cache misses.

Chapter 4

Binary Spatial Partitioning for Domain-Mapping

In this chapter, we are primarily interested in the data parallelization where the scene database (scene description with the auxiliary data structure) as well as computations are distributed fairly among processors of the multicomputer, since the whole database may not fit into the local memories of the processors. The rendering method assumed is the ray casting, a similar technique to ray tracing that ignores shadow and other secondary rays, and the scene rendered is assumed to contain many objects that do not fit into the local memory of a single processor. In an efficient data parallelization, the subdivision of object space and mapping to the processors play an important role, and thus should be carefully applied.

The subdivision and mapping should be performed in such a way that each processor is assigned almost equal amount of storage and computational load. Furthermore, the near objects should be kept in the local memory of near processors to achieve better graphical coherence such as *data coherence* [12]. When the scene database is decomposed using spatial subdivision of 3-D space, it is almost unavoidable to split and duplicate some of the objects, called *shared objects*, in the local memories of processors. This is due to the fact that the chosen optimal splitting planes will most probably intersect some

of the objects of the complex scene. The proposed subdivision algorithm has efficient data structures to locate the splitting planes which result in less splitted and duplicated objects. Another advantage of the algorithm is that the mapping of object data and computations to processors is achieved during the subdivision process.

Although the proposed spatial subdivision algorithm is analyzed and implemented for parallel ray casting, it can also be used for other scan-conversion rendering algorithms and the parallel ray tracing using a data parallelization scheme with a caching mechanism. Particularly, it will be valuable for the initial assignment of scene database and the associated computations to processors.

In the following section, several spatial subdivision schemes for object-space decomposition problem are described briefly. Next, we propose a spatial subdivision algorithm for parallel ray casting and present a graph-based approach to improve the mapping of resultant parts to the processors.

4.1 Object-Space Subdivision

Data parallelism requires decomposition and distribution of the scene database. The techniques developed to improve the naive ray tracing algorithm can be utilized in the decomposition of the scene database. These techniques are *hierarchy of bounding volumes* [11] and *spatial subdivision* [10, 16] and can be adapted to parallel ray tracing as follows. The first technique forms a hierarchy of clusters consisting of neighboring objects. In the parallel processing case there might be two approaches, namely *static* and *on-demand*, to accomplish a fair distribution of computations and storage. The former approach performs a static allocation by partitioning the entire hierarchy into a set of clusters each of which is assigned to a node processor. This resembles a graph partitioning process [15]. The latter approach allocates object space data and relevant computations to the node processors on-demand. The second technique, called spatial subdivision, decomposes the 3-D space containing the scene into disjoint rectangular prisms. As in the first technique,

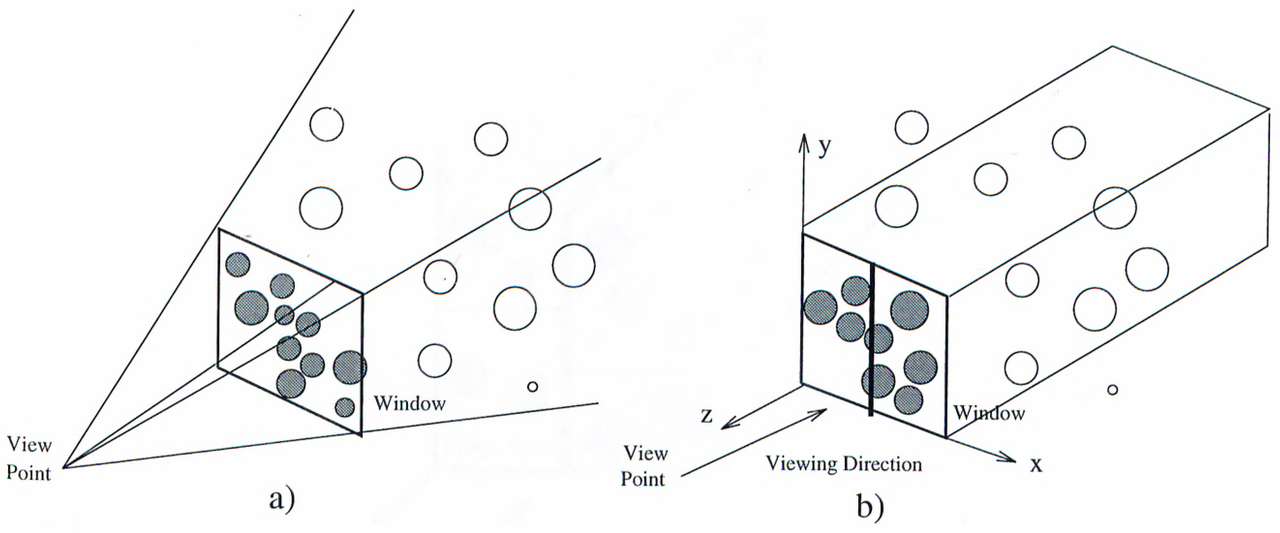


Figure 4.1: Viewing Volumes with (a) pyramid (b) rectangular shapes.

the resulting prisms are distributed to the node processors either statically or on-demand [12, 19, 26]. In this research, the second technique, spatial subdivision, is preferred to decompose the object space due to its use of spatial coherence.

4.2 Binary Space Partitioning for Parallel Ray Tracing

Although both octree and regular subdivision schemes have very nice properties when used in conventional ray tracing algorithm, it is difficult to achieve computational load balance among processors, if some coherence properties such as object, data, and image coherence are to be utilized. A manifestation of coherence called *data coherence* (first exploited by Green and Paddon [12]) is a very powerful and useful property that might reduce the communication overhead. Communication among the node processors is one of the most time consuming operations in an object-space parallel ray tracing system. Therefore, exploiting data coherence is essential in speeding up object-space parallel ray tracing. In order to exploit data coherence, we propose a variant of BSP. We call it BBSP (Balanced Binary Space Partitioning) since a complete binary tree is generated

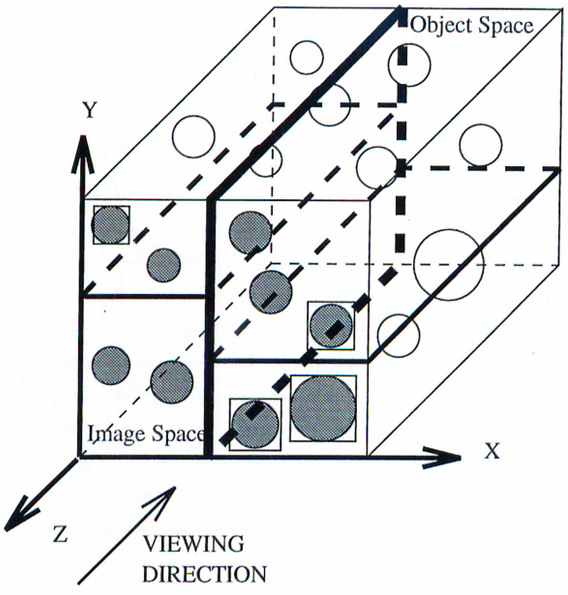


Figure 4.2: Decomposition of a rectangular region and resultant 3-D volumes.

at the end of the subdivision. The subdivision is carried out on a window defined over a viewing plane onto which the objects in the scene are projected (parallel) (Figure 4.1). The subdivision produces a set of disjoint rectangular regions on the window and a set of 3-D volumes obtained by extending the rectangular regions in the viewing direction. By means of this subdivision preprocess, the decomposition of both object space data describing the scene and the image-space computations associated with the pixels on the window are performed. In a general case, some of the objects defined in a scene may not project on a given window or some may be outside the viewing volume. In such cases, it is not possible to decompose both object space and image space as just mentioned. Therefore we assume that the objects lie within the viewing volume and a given scene is visualized from a distant point so that the rays are parallel to each other. Under the given assumptions, the viewing volume and the produced 3-D volumes have rectangular (parallelepiped) shape rather than pyramid shape.

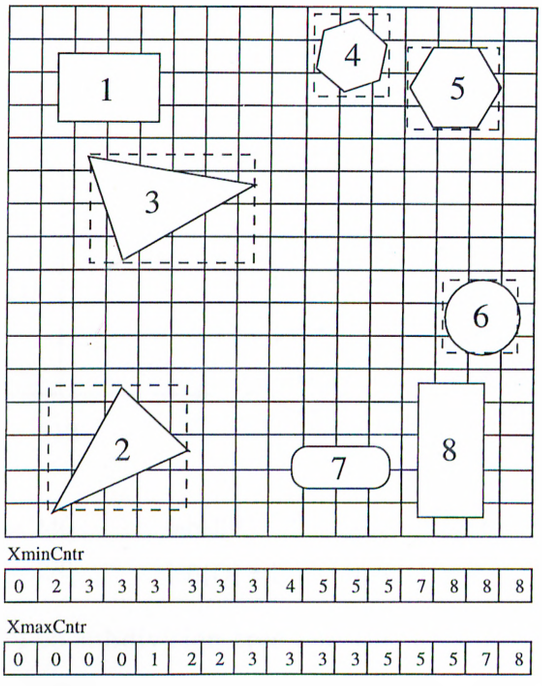


Figure 4.3: A sample scene projected onto the viewing plane. Here, XminCntr and XmaxCntr contain values after the prefix sum operation.

4.3 Balanced Binary Space Partitioning Algorithm

Before executing the main loop of the BBSP algorithm, all objects in the view volume are projected onto the window that is the initial rectangular region to be subdivided into two rectangular subregions. The projections of the objects are then surrounded by bounding boxes to simplify the computations as seen in Figure 4.3. After this operation, each object in the scene has four parameters: $xmin, xmax, ymin$ and $ymax$. Here, $xmin$ and $xmax$ are the left and the right borders of the bounding box, respectively. Similarly, $ymin$ and $ymax$ are used to hold the bottom and the top borders of the bounding box.

In the main loop of the algorithm, each generated rectangular subregion is subdivided into two subsubregions by a splitting plane which is parallel to either x-z (horizontal) or y-z (vertical) plane as shown in Figure 4.2. This subdivision process proceeds in a

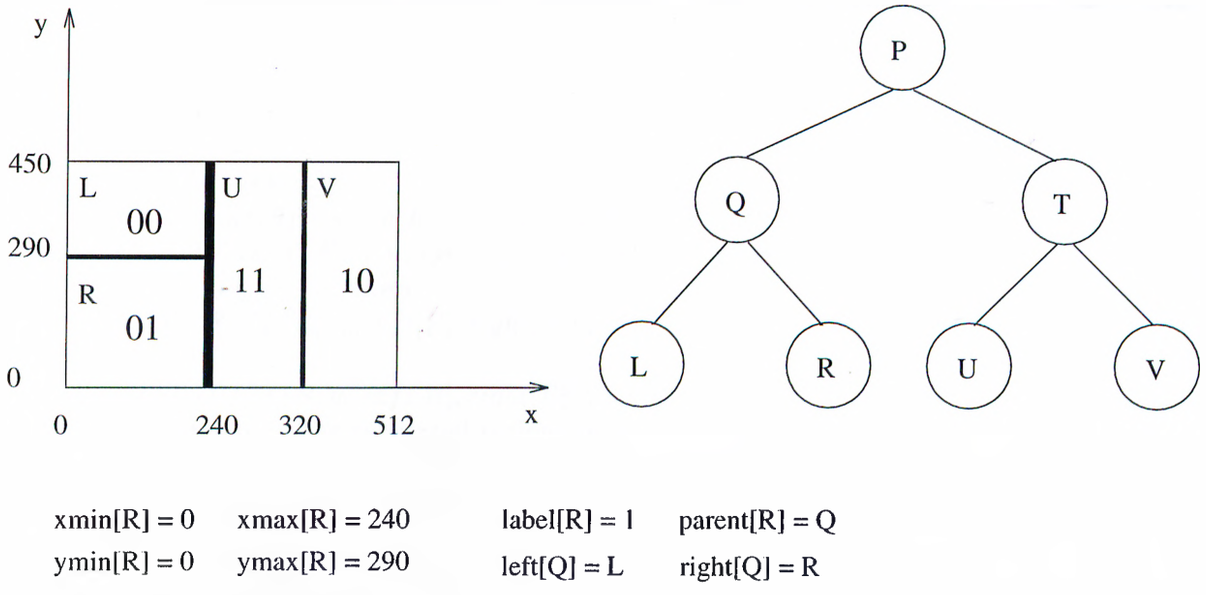


Figure 4.4: The subdivision tree and the attributes of region R.

breadth-first manner until the number of generated subregions (the number of leaves of the generated tree) becomes equal to the number of processors P. Here, the number of processors is assumed to be a power of two.

The basic algorithm is presented in pseudo-code in Figure 4.5. In the description of the algorithm, a rectangular region is denoted by letter R which has a number of attributes. Round brackets (or parentheses) are used to enclose parameters of a procedure or a function or to enclose the indices of an array to access an array element. Square brackets, on the other hand, are used to access attributes of a variable. For instance, $xmax[R]$ represents $xmax$ attribute (field) of region R. $VCOST(R, m, n, x_b)$ is a function with parameters R, m, n and x_b. All attributes of a region, the procedures and the functions used will be described in the following paragraphs in detail.

A rectangular region is the basic entity on which the major operations are applied. The results of the operations are stored in the attributes of the rectangular regions. The following attributes are used to store the information about the rectangular regions: $xmin[R]$, $xmax[R]$, $ymin[R]$ and $ymax[R]$ are left, right, bottom and top borders of rectangular region R. The number of objects intersecting a region R is denoted by

```
BBSP(W, P)
    Enqueue(Q, W)
    repeat
        R ← Dequeue(Q) ;
        m ← ymax[R] − ymin[R] ; n ← xmax[R] − xmin[R] ;
        C_v ← VCOST(R, m, n, x_b) ; C_h ← HCOST(R, m, n, y_b) ;
        if C_v/m ≤ C_h/n then
            VSPLIT(R, x_b, left[R], right[R]) ;
        else
            HSPLIT(R, y_b, left[R], right[R]) ;
        /* mapping of the generated regions to processors */
        if R = right[parent[R]] then
            label[left[R]] = label[R] || "1" ; label[right[R]] = label[R] || "0" ;
        else
            label[left[R]] = label[R] || "0" ; label[right[R]] = label[R] || "1" ;
        Enqueue(Q, left[R]) ; Enqueue(Q, right[R]) ;
    until number of regions = P ;
```

Figure 4.5: The main body of the proposed BBSP algorithm.

```
VCOST(R, m, n, x_b)
    mincost = ∞ ;
    for b = 0 to n do
        L_b = XminCntr[R](b) ;
        S_b = XminCntr[R](b) − XmaxCntr[R](b) ;
        R_b = number[R] − L_b + S_b ;
        cost = |b × L_b − (n − b) × R_b| + n × S ;
        if cost < mincost then
            mincost = cost ;   b_min = b ;
    x_b = b_min ;
    return mincost ;
```

Figure 4.6: Function to find the optimal vertical splitting plane and compute its cost.

$number[R]$. The label of the processor onto which the region R is to be mapped is given by $label[R]$. The left and the right children of the node in the BBSP tree corresponding

VSPLIT$(R, x_b, left[R], right[R])$

$b = x_b$;
$S_b = XminCntr[R](b) - XmaxCntr[R](b)$;
while $S_b \neq 0$ **do**
  for each *object* $o \in XminObj[R](b)$ **do**
    if $xmax[o] \geq x_b$ **then**
      $xmin[o'] = x_b$; $xmax[o'] = xmax[o]$;
      $ymin[o'] = ymin[o]$; $ymax[o'] = ymax[o]$;
      $XminObj[R](x_b) = XminObj[R](x_b) \cup \{o'\}$;
      $XminCntr[R](x_b) = XminCntr[R](x_b) + 1$;
      $xmax[o] = x_b - 1$;
      $XmaxCntr[R](x_b - 1) = XmaxCntr[R](x_b - 1) + 1$;
      $S_b = S_b - 1$;
  $b = b - 1$;
/* create left and right subregions of the parent region R */
$xmin[left[R]] = xmin[R]$; $xmax[left[R]] = x_b - 1$;
$ymin[left[R]] = ymin[R]$; $ymax[left[R]] = ymax[R]$;
$XminCntr[left[R]](0 \ldots x_b - 1) \leftarrow XminCntr[R](0 \ldots x_b - 1)$;
$XmaxCntr[left[R]](0 \ldots x_b - 1) \leftarrow XmaxCntr[R](0 \ldots x_b - 1)$;
$XminObj[left[R]](0 \ldots x_b - 1) \leftarrow XminObj[R](0 \ldots x_b - 1)$;
$xmin[right[R]] = x_b$; $xmax[right[R]] = xmax[R]$;
$ymin[right[R]] = ymin[R]$; $ymax[right[R]] = ymax[R]$;
$XminCntr[right[R]](x_b \ldots n) \leftarrow XminCntr[R](x_b \ldots n) - XminCntr[R](x_b - 1)$;
$XmaxCntr[right[R]](x_b \ldots n) \leftarrow XmaxCntr[R](x_b \ldots n) - XmaxCntr[R](x_b - 1)$;
$XminObj[right[R]](x_b \ldots n) \leftarrow XminObj[R](x_b \ldots n)$;
$YCONSTRUCT(left[R])$;
$YCONSTRUCT(right[R])$;

YCONSTRUCT(R)

for each *object* $o \in XminObj[R]$ **do**
  $lmin = ymin[o] - ymin[R]$;
  $lmax = ymax[o] - ymin[R]$;
  $YminCntr[R](lmin) = YminCntr[R](lmin) + 1$;
  $YmaxCntr[R](lmin) = YmaxCntr[R](lmin) + 1$;
  $YminObj[R](lmin) = YminObj[R](lmin) \cup \{o\}$;

Figure 4.7: VSPLIT is a procedure to split a given region with the associated data structures vertically. YCOSTRUCT is used to construct y-direction data structures.

to the region R are denoted by $left[R]$ and $right[R]$, respectively. Finally, $parent[R]$ is the parent node corresponding to the region from which R is obtained. As shown in Figure 4.4, the left and the right children of a subdivided parent region are the nodes corresponding to the regions which are to the left (or above) and to the right (or below) of the splitting plane.

In addition to the primitive attributes previously introduced, each region is associated with the following complex attributes for both x and y dimensions: $XminCntr[R]$, $XmaxCntr[R]$, $XminObj[R]$ for the x dimension and $YminCntr[R]$, $YmaxCntr[R]$, $YminObj[R]$ for the y dimension. $XminCntr$ and $XmaxCntr$ are two integer arrays defined to hold the information related to the distribution of objects along the x dimension. $XminObj$ is a pointer array where each element points to a list of objects according to the $xmin$ of the objects. The $YminCntr$, $YmaxCntr$ and $YminObj$ are similar data structures constructed and used for the y dimension. Assuming that a given region consists of $m \times n$ pixels (resolution), the arrays for x and y dimensions have size of n and m, respectively.

To be more specific, $XminCntr[R](b)$ that is associated with region R contains the number of objects whose $xmin$ is equal to b for $b = 1, \ldots, n$; $XmaxCntr[R](b)$ of R contains the number of objects that have $xmax = b$ for $b = 1, \ldots, n$. $XminObj[R](b)$ associated with region R points to a list of objects that have $xmin = b$ for $b = 1, \ldots, n$. Similar examples can also be given for the arrays in the y dimension of rectangular region R.

The basic algorithm given in Figure 4.5 consists of a single loop which is executed until the number of generated rectangular regions is equal to the number of processors, P. The Q, a variable of type queue, contains the generated rectangular regions. At the beginning of the algorithm, variable Q is initialized with the window onto which the objects are projected and from which other rectangles are generated. Through each *pass* of the loop, a rectangle is picked up from Q using *Dequeue*, it is subdivided and the resultant two rectangles are appended to the Q, using *Enqueue*, replacing the parent region. The statements and the functions within the loop performs all operations regarding the

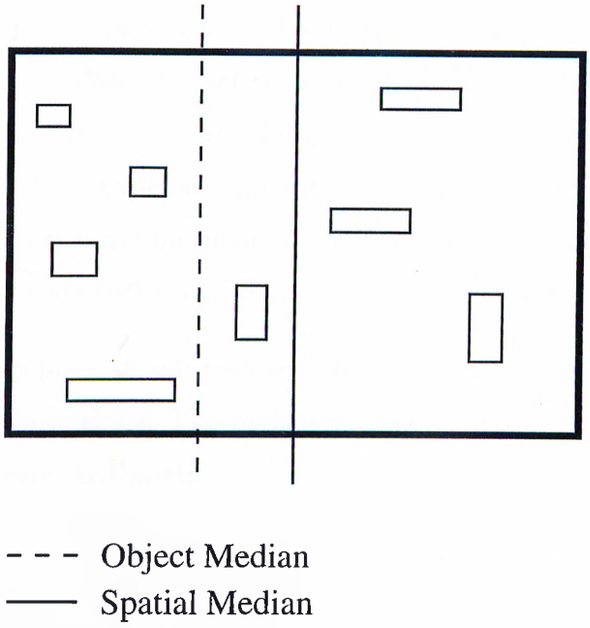


Figure 4.8: Choosing the location of the splitting plane.

subdivision operation such as computing the cost of a splitting plane, locating the optimal splitting plane, splitting a region, labeling of the regions and initializing the fields of the regions appropriately. To perform these operations, the algorithm incorporates three major operations: VCOST and HCOST (Figure 4.6) to compute the cost of a splitting plane; VSPLIT and HSPLIT (Figure 4.7) to split a given rectangular region; the second *if* statement to label the generated regions.

4.3.1 Identifying Optimal Splitting Planes

A splitting plane divides a given rectangular region of the screen into two disjoint rectangular subregions consisting of pixels. A splitting plane is characterized by its cost, direction (vertical or horizontal) and its position where the screen is cut.

In BSP trees, the location of the splitting plane is usually chosen along either object median or spatial median as in Figure 4.8. MacDonald and Booth [21] have examined two heuristics for space subdivision using BSP. They pointed out that the probability of intersection of a given ray with an object is proportional to the surface area of the

object - called the *surface area heuristic.* Using this heuristic, they have also found out that the optimal splitting plane lies between the object median and the spatial median. This result reduces the required search range to find out the location of the splitting plane. However, it is still an expensive operation to carry out search within the reduced search range since the cost function involves floating point arithmetic. Furthermore, the analysis in [21] neglects the existence of shared objects between the generated subregions.

In this work, we propose an efficient search algorithm for finding optimal splitting planes during space subdivisions. The proposed algorithm uses efficient data structures and requires only integer arithmetic. In the proposed algorithm, the position of an optimal splitting plane is determined by using an objective function that considers both the minimization of the computational load-imbalance and the number of shared objects between the generated subregions. The proposed objective function exploits the surface area heuristic for maintaining the computational load balance between the generated subregions.

The functions VCOST and HCOST find the positions of the optimal splitting planes in vertical and horizontal directions, respectively, and compute their associated cost. These two functions accept four parameters: the region to be subdivided R, the height and the width (resolution) of the rectangular region (m and n) and the position of the splitting plane (x_b or y_b). Only the function VCOST has been presented in order to save space. The function HCOST can be obtained by replacing X's by Y's. As shown in the pseudo-code of the function VCOST, for all possible splitting planes a cost computation is performed and the position of the splitting plane with the minimum cost in each direction (vertical or horizontal) is returned as the result.

The cost of a vertical splitting plane b on a region R consisting of $n \times m$ pixels (resolution) is defined as

$$C_v(b) = \frac{|m \times b \times L_b - m \times (n-b) \times R_b|}{n \times m \times N} + \frac{S_b}{N} \tag{4.1}$$

for $b = 1, \ldots, n$, where N denotes the total number of objects projected onto the viewport under consideration. The objective function for a horizontal splitting plane can easily be

obtained by exchanging n with m in Equation 4.1. Here, L_b and R_b denote the number of objects in the $left[R]$ and $right[R]$, respectively. Furthermore, S_b denotes the number of shared objects straddling across the splitting plane b.

The denominator of the first term in Equation 4.1 denotes the total computational load associated with the window when only primary rays are considered. Hence, the first term in Equation 4.1 represents percent load imbalance between the two subregions generated by a particular splitting plane. Similarly, the second term in Equation 4.1 denotes percent number of shared objects between those two subregions. The shared objects cause several problems. First, the shared objects are duplicated in the local memories of the processors to which these objects are assigned. Second, an intersection test with a shared object might be repeated if the first intersection point is not inside the subvolume that is assigned to the processor performing the test. As in conventional ray tracing, there might be another closer intersection point within the next subvolume along the path of the ray.

The objective function in Equation 4.1 is computed for all splitting planes in both vertical and horizontal directions. The splitting plane with the smallest cost is chosen as the optimal splitting plane. Hence, the objective function should be efficiently computed. The objective function for vertical splitting planes can be simplified as:

$$C_v(b) = \frac{1}{n \times N}\{|b \times L_b - (n-b) \times R_b| + n \times S_b\} \tag{4.2}$$

for $b = 1, \ldots, n$. The simplification for horizontal splitting planes can be obtained by replacing n with m in Equation 4.2. The parameter $1/N$ can be neglected since it is a constant factor common in all cost computations (both vertical and horizontal). Similarly, the parameters $1/m$ and $1/n$ appear as constant factors common in vertical and horizontal splitting plane computations, respectively. Hence, it is sufficient to compute the following functions.

$$C_v(b) = |\{b \times L_b + (n-b) \times R_b\}| + n \times S_b \tag{4.3}$$

$$C_h(b) = |\{b \times L_b + (m-b) \times R_b)\}| + m \times S_b \tag{4.4}$$

for $b = 1, \ldots, n$ and $b = 1, \ldots, m$ in order to find the optimal vertical and horizontal splitting planes b_v^{min} and b_h^{min}, respectively. The optimal splitting plane is then chosen

among these two splitting planes by comparing $C_v(b_v^{min})/m$ with $C_h(b_h^{min})/n$. This formulation enables the use of only integer arithmetic during the cost computations.

As stated before, four major data structures exist to hold the distribution of objects along vertical and horizontal directions: *XminCntr* and *XmaxCntr*, where *XminCntr[R](b)* and *XmaxCntr[R](b)* contain the number of objects whose *xmin* and *xmax* values are equal to b for region R, respectively, for $b=1,2,\ldots,n$. The *YminCntr* and *YmaxCntr* are similar data structures constructed and used for the y dimension.

Having formed these data structures as explained before, prefix sum operation is performed on these integer arrays. These integer arrays are then used in the computation of the objective functions in Equation 4.3 and 4.4. These equations need the values of R_b, L_b and S_b for each possible splitting position b. After prefix sum operations, *XminCntr[R](b)* and *XmaxCntr[R](b)* contain the number of objects whose *xmin* and *xmax* values are equal to or less than b in region R, respectively. Hence, *XminCntr[R](b) (YminCntr[R](b))* denotes the number L_b of objects in the left (bottom) subregion of the vertical (horizontal) splitting plane. Similarly, *XmaxCntr[R](b) (YmaxCntr[R](b))* denotes the number of objects in the left (bottom) subregion which do not straddle across the vertical (horizontal) splitting plane b. Hence, S_b and R_b can easily be computed as

$$S_b = L_b - XmaxCntr[R](b) \tag{4.5}$$

$$R_b = (N + S_b) - L_b \tag{4.6}$$

for a vertical splitting plane b. For a horizontal splitting plane b, S_b and R_b can similarly be computed using these two equations by replacing *XmaxCntr* in Equation 4.6 with *YmaxCntr*. Note that the values of R_b, L_b and S_b are efficiently computed using only 3 integer additions which will be performed for all possible splitting planes.

4.3.2 Splitting

Having determined the optimal splitting plane, the region R is splitted into two using VSPLIT or HSPLIT which accept four parameters. The first parameter is the rectangular region R to be subdivided. The second parameter is the position

of the splitting plane. The last two parameters are the left ($left[R]$) and the right ($right[R]$) children generated after splitting the region R. Essentially, these procedures generate two new rectangular regions with their fields (attributes) initialized. Note that $number[left[R]] + number[right[R]] \geq number[R]$ which means that the sum of the objects associated with the resultant rectangular regions might be greater than or equal to the total number of objects in the subdivided parent rectangular region. This difference is due to the duplicated objects, called shared objects, intersected by the splitting plane. The variable o' used throughout the VSPLIT function represents an object created from a shared object intersecting a splitting plane. The appropriate updates due to the created objects are applied to certain data structures of each generated region. When splitting a region in the x dimension (vertically), it is easier to form the x-dimension data structures than y-dimension data structures. The x-dimension data structures are simply found out as if the arrays are cut at the splitting plane position with slight modifications. On the other hand, the y-dimension data structures are formed using the x-dimension data structures by starting from scratch. YCONSTRUCT constructs the y-dimension data structures of the generated regions using x-dimension data structures. Similarly, when a region is splitted in the y-dimension, the x-dimension data structures are formed using XCONSTRUCT which can be obtained by replacing Y's and X's with each other in the YCONSTRUCT.

4.3.3 Assignment of Generated Regions to Processors

The proposed algorithm achieves the mapping of the generated subregions during the recursive subdivision process. Each generated subregion is assigned a label that corresponds to the processor-group to which it is assigned. Initially, the window W is assumed to be assigned to all processors in the parallel architecture. While splitting a region into two subregions, the processor-group assigned to that region is also split into two halves and these two halves are assigned those two subregions, respectively. This recursive spatial subdivision of the window proceeds together with the recursive subdivision of the processor interconnection topology. The recursive subdivision and

assignment scheme to be adopted for the processor interconnection topology is a crucial factor in achieving the data coherence mentioned earlier.

In this work, we propose a recursive labeling scheme for the generated regions during the recursive subdivision of the window. This labeling scheme emulates the recursive definition of the hypercube interconnection topology as the target architecture for the object-space parallel ray tracing algorithm. However, the proposed labeling can easily be adopted to other parallel architectures implementing symmetric and recursive interconnection topologies (e.g., 2-D Mesh and 3-D Mesh) with minor modifications.

Here, we will briefly summarize the topological properties of hypercubes exploited in the proposed labeling. A multicomputer implementing the hypercube interconnection topology consists of $P = 2^d$ processors with each processor being directly connected to d other neighbor processors. In a d-dimensional hypercube, each processor can be labeled with a d-bit binary number such that the binary label of each processor differs from its neighbor in exactly one bit. A *channel* c defines the set of $P/2$ links connecting neighbor processors whose binary labels differ only in bit c, for $c = 0, 1, 2, \ldots, d-1$. In the recursive definition of the hypercube topology, a d-dimensional hypercube is constructed by connecting the processors of two $(d-1)$-dimensional hypercube in a one-to-one manner. Hence, a d-dimensional hypercube can be subdivided into two disjoint $(d-1)$-dimensional hypercubes, called subcubes, by *tearing* the hypercube across a particular channel (e.g., $c = d-1$). Each one of these two $(d-1)$-dimensional subcubes can in turn be divided into two disjoint $(d-2)$-dimensional subcubes by tearing them across another channel (e.g., $c = d-2$). Hence, d such successive tearings along different channels (e.g., $c = d-1, d-2, \ldots, 1, 0$) result in 2^d 0-dimensional subcubes (i.e., processors). An h-dimensional subcube in a d-dimensional hypercube ($0 \leq h \leq d$) can be represented by a d-tuplet containing h *free-coordinates* (x's) and $d-h$ fixed-coordinates (0's and 1's) [24]. In the proposed mapping scheme, the label Q of the initial rectangular region (window W) is initialized to null. Consider the subdivision of a particular subregion labeled as Q by a vertical or horizontal splitting plane. Note that the label Q of this subregion is a q-bit binary number where q denotes the depth of this subregion in the subdivision recursion

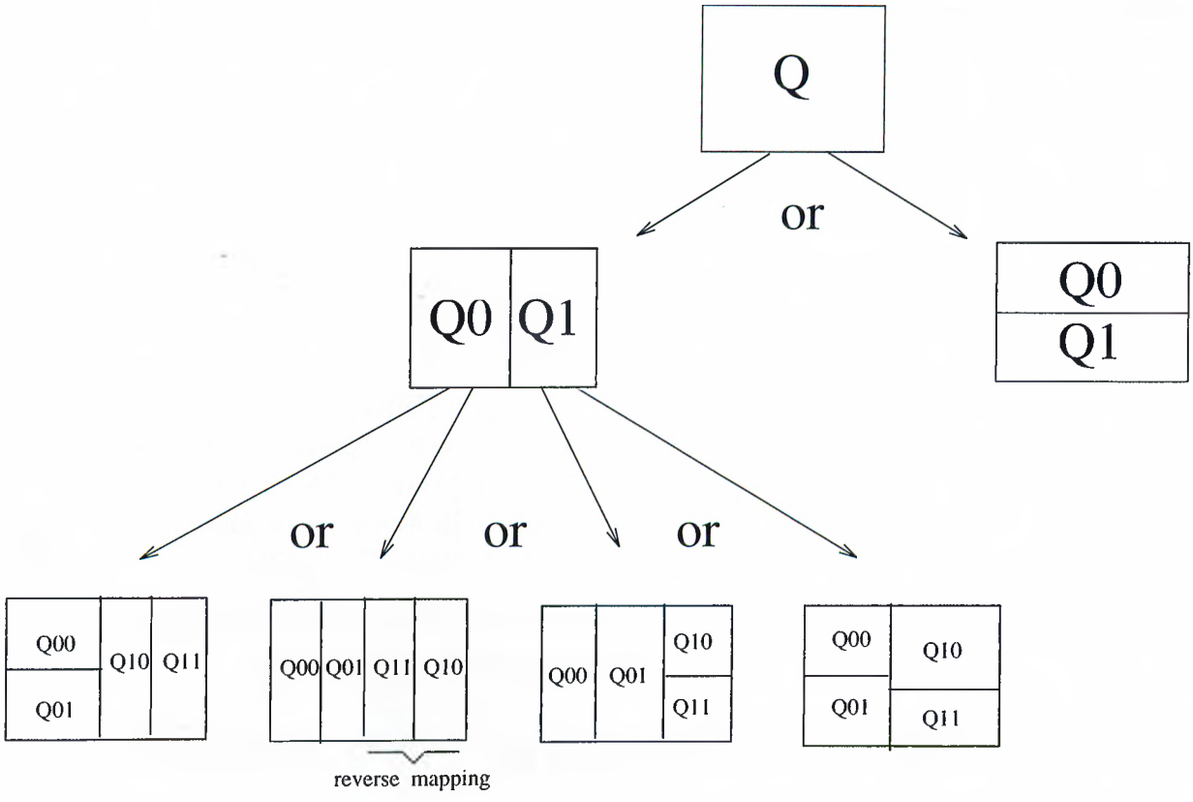


Figure 4.9: Labeling of the generated regions using Gray code ordering.

tree. Hence, subregion Q is already mapped to the $(d-q)$-dimensional subcube Qx...x. The left (below) and right (above) subsubregions generated by a vertical (horizontal) splitting plane are labeled as $Q0$ and $Q1$, respectively. This labeling corresponds to tearing the subcube Qx...x across channel $d-q-1$ and mapping the resulting $(d-q-1)$-dimensional subsubcubes Q0x...x and Q1x...x to left (below) and right (above) subsubregions, respectively (see Figure 4.9). However, if two subregions $Q0$ and $Q1$ generated from the same region by a vertical (horizontal) splitting plane are both splitted again by vertical (horizontal) planes, then the subsubcube-to-subsubregion assignment in one of these two subregions is performed in reverse order. The proposed labeling scheme tries to maximize the data coherence by mapping neighboring subregions to neighboring subcubes, as much as possible, during the recursive subdivision process. Figure 4.9 illustrates the possible labeling combinations in a particular subpath of the recursion tree.

```
NEIGHBOR_FIND
  for each region R do
    XmaxReg(xmax[R]) ← XmaxReg(xmax[R]) ∪ {R} ;
    YmaxReg(ymax[R]) ← YmaxReg(ymax[R]) ∪ {R} ;
  for each region R do
    for each region Q(≠ R) ∈ XmaxReg(xmin[R]) do
      if OVERLAP(ymin[R], ymax[R], ymin[Q], ymax[Q]) then
        West[R] ← West[R] ∪ {Q} ;
        East[Q] ← East[Q] ∪ {R} ;
    for each region Q(≠ R) ∈ YmaxReg(ymin[R]) do
      if OVERLAP(xmin[R], xmax[R], xmin[Q], xmax[Q]) then
        South[R] ← South[R] ∪ {Q} ;
        North[Q] ← North[Q] ∪ {R} ;
```

Figure 4.10: Neighbor finding algorithm for BBSP.

4.4 Neighbor-Finding Algorithm for BBSP

In ray tracing and other graphics algorithms using 3-D spatial subdivision, there is a need to know the *adjacent* regions of a currently processed region. The adjacent regions are termed as *neighbors*. Neighbor-finding techniques in quadtree and octree are extensively studied in the literature [30]. In object-space parallel ray tracing, a processor that does not find any intersection within the volume assigned to itself should know which neighbor region is the next region on the path of the ray. After determining the next neighbor region through which the ray passes, either the object information in that region is requested or the ray passes to that region to carry out the necessary computations.

Since the subdivision process performed here does not have any regular pattern and thus generates regions in any size and position, neighbor-finding algorithm is quite different from those used in quadtree and octree. Neighbor-finding algorithm developed here is executed only once after all the regions are generated. At the end of the neighbor-finding algorithm, each processor owns four lists to hold *West*, *East*, *South* and *North* neighbor regions which are then used during the execution of the program such as ray

tracing loop. The memory occupied to hold the neighborhood information is negligible compared to the local memory of a node processor. Therefore, storing neighborhood information is reasonable in terms of both storage and time.

The proposed neighbor-finding algorithm deals with the problem in the discrete space that does not conflict with the subdivision algorithm proposed before. The algorithm incorporates a single type of data structure which is an array of pointers. For each dimension, one such array of pointers is used to point to the regions according to their boundaries: $XmaxReg$ and $YmaxReg$ are used to index the regions according to their right and top boundaries, respectively. For example, $XmaxReg(b)$ ($YmaxReg(c)$) points to a list of all regions that have the right (top) boundary equal to b (c) for $b = 1...n$ ($c = 1...m$) where $n \times m$ is the resolution of the root region (viewport) from which the resultant regions are generated. Note that the right, the left, the top and the bottom boundaries of a region R can be accessed using the attributes $xmax[R]$, $xmin[R]$, $ymax[R]$ and $ymin[R]$, respectively.

At the beginning of the algorithm as given in Figure 4.10, the four arrays are initialized using the boundaries of all regions. Then, for each region, the $West$ and the $South$ neighbors are determined using the mentioned arrays. To find the $West$ neighbor of a region R, all regions with the right boundaries equal to the left boundary of R are retrieved using the $XmaxReg(xmin[R])$. Since only some of these retrieved regions could be west neighbors of R, it is necessary to test which regions' right boundaries overlap in the y dimension with the left boundary of R. This test can simply be accomplished through searching and comparing the boundaries using the OVERLAP function. It is obvious that west neighbors of a region should have that region as one of their east neighbors.

It can be shown that the asymptotic complexity of this algorithm is $O(P^2)$, where P is equal to the number of regions. When P is not too large (up to 128), the time will be insignificant since the algorithm runs in the preprocessing step. It is possible to make the algorithm more efficient if each list pointed by the mentioned array elements is sorted. Then the search will be within a reduced range of the sorted list.

4.5 A Graph Based Approach to Improve the Mapping

In the given BBSP method, the mapping of generated regions to the processors is performed during the subdivision process. In order to evaluate and improve the mapping process, the mapping problem is converted into a graph partitioning problem to be solved using the Kernighan-Lin heuristic.

The result of the BBSP will be a set of regions constituting a *floorplan* F which is defined as a dissection of a rectilinear region (window) W into a collection of rectangles. Each rectangle of the floorplan represents a subvolume that is obtained by extending the corresponding subregion of the window W in the z dimension. When the subdivided region W is rectangle then F is called as a rectangular floorplan. Each rectangle in the floorplan has at least one neighbor.

A *rectangular dual graph*, also called *rectangle adjacency graph*, of a rectangular floorplan F is a planar graph. $G = (R, E)$, where $R = \{R_1, ..., R_2\}$ the set of rectangles and $(R_i, R_j) \in E$ if and only if the regions R_i and R_j are adjacent to each other in the floorplan F.

In the following sections, the improvement of mapping is handled in two major stages: in the first stage, the scene is represented by a rectangle adjacency graph G obtained from the rectangular floorplan F. In the second stage, the nodes of the rectangle adjacency graph are mapped to the processors of the multicomputer. The two stages are discussed in detail in the next sections.

4.5.1 Generation of Rectangle Adjacency Graph

This step takes a rectangular floorplan F as input generated from the partitioning of the projection plane (window W.) The output generated at the end of this step is a graph that will be input to the second step. The graph generated is a rectangle adjacency

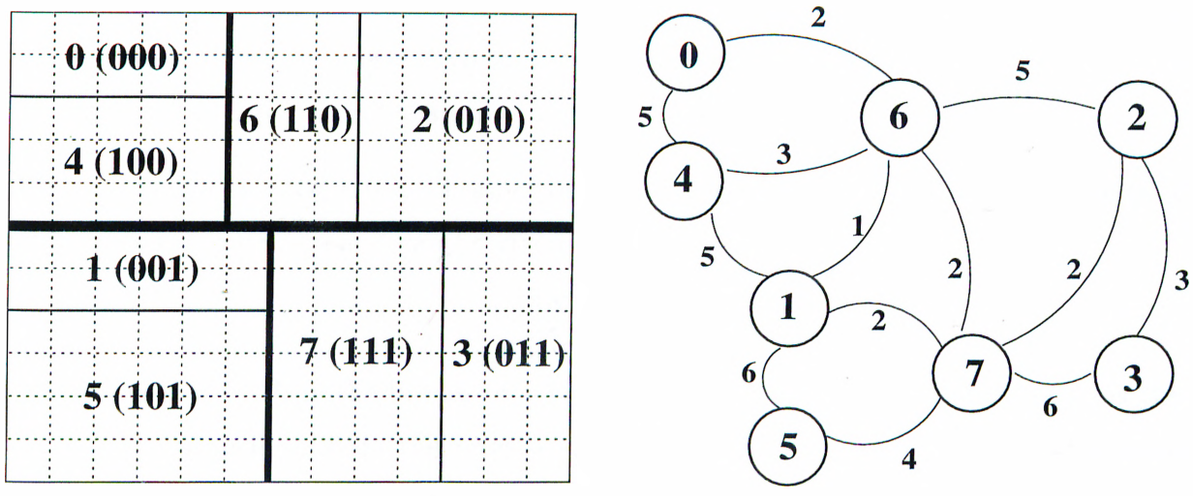


Figure 4.11: Generating a rectangle adjacency graph from a rectangular floorplan.

graph, where each node represents a rectangular region and the nodes are connected by edges according to adjacency relation. For example, the node U is connected to V if and only if the associated regions of these nodes are adjacent to each other. An edge between any two nodes of the graph is given a weight that is computed as the length of the border common to both rectangular regions associated with the nodes connected by that edge. This weight is proportional to the surface area between two 3-D volumes which are obtained by extending the respective 2-D rectangular regions represented by the nodes of the graph. Figure 4.11 includes a sample scene and a corresponding graph.

4.5.2 One-to-one Mapping

The input to this stage of the algorithm is a graph obtained in the first step. A one-to-one mapping of P generated nodes onto a multiprocessor with P processors can be performed in $P!$ different ways. The goal is to reduce this exponential search space through using an efficient heuristic in order to optimize the communication cost between processor nodes. The graph-based mapping algorithm used here is an iterative improvement heuristic that is conceptually similar to the Kernighan-Lin (KL) algorithm [2]. The algorithm proposed by Kernighan and Lin is a popular non-trivial heuristic to solve the graph

partitioning problem [17]. This algorithm provides hill-climbing ability to strictly local search methods by finding a favorable sequence of swaps of nodes between partitions rather than a single favorable swap of groups of nodes.

The total expected interprocessor communication cost depends on the mapping of regions onto processors and can be given as:

$$C = \sum_{i<j} d_{M_i,M_j} w_{i,j} \tag{4.7}$$

In Equation 4.7, M_i is the processor node on which the region i resides. d_{M_i,M_j} is the hamming distance between the processors M_i and M_j. $w_{i,j}$ is the weight of the edge connecting the nodes of the graph corresponding to the regions i and j.

The algorithm looks for all pairs of regions that maximally decrease the total interprocessor communication cost when they are swapped. Swapping of two regions means to interchange the assigned processors of these swapped regions. Total decrease in the interprocessor communication cost corresponds to the *gain* associated with the swapping of two regions.

The KL algorithm is presented using pseudo code in Figure 4.12. Here, R is the set of rectangular regions corresponding to the nodes of rectangular adjacency graph. U contains unlocked nodes of the graph. G holds the gains $g_{i,j}$ associated with the swapping of the rectangles i and j that reside on the processors M_i and M_j, respectively. D is the second graph indicating the interconnection topology of the processors. The hamming distance between processors i and j (d_{M_i,M_j}) is obtained from this graph. H is a priority queue which can be implemented using a binary heap. It is used to keep the gains in sorted order for efficient access.

At the beginning of the outer loop of the algorithm, all rectangles R are put into U from which the swaps will be selected. Then the gains of all possible swaps between any two rectangles are computed and stored in G. When any two rectangles are swapped (interchange the processors assigned), the total expected communication cost C given in Equation 4.7 may increase, decrease or remain the same. This difference of cost is called *gain* which is positive for cost decrease and, negative for cost increase.

```
/* KL one-to-one Mapping */
MAIN_KL
    while g_max ≥ 0 do
        U ← R ;
        G ← INIT_GAIN(D) ;
        insert all swaps into H ;
        for s = 1 to P/2 do
            extract the region swap (i, j) with maximum gain from H ;
            swap_gains(s) ← g_{i,j} ;
            U ← U − {i} − {j} ;
            DELETE(H, i) ;
            DELETE(H, j) ;
            lock regions i and j ;
            UPDATE_GAIN(D, U, i, j, G) ;
            UPDATE_GAIN(D, U, j, i, G) ;
        perform prefix sum on array swap_gains(1..P/2) ;
        find s_max with maximum g_max = swap_gains(s_max) ;
        if g_max > 0 then
          realize first s_max swaps ;

UPDATE_GAIN(D, U, i, j, G)
    for each unlocked region p ∈ Adj[i] do
        for each unlocked region q(≠ p) ∈ U do
            g_{p,q} = g_{p,q} + w_{i,p}[(d_{M_p,M_j} − d_{M_p,M_i}) + (d_{M_q,M_i} − d_{M_q,M_j})] ;

INIT_GAIN(D)
    for i = 1 to P do
        for j = i + 1 to P do
            g_{i,j} ← 0 ;
            for each region p ∈ Adj[i] do
                g_{i,j} = g_{i,j} + w_{i,p}(d_{M_i,M_p} − d_{M_j,M_p}) ;
            for each region q ∈ Adj[j] do
                g_{i,j} = g_{i,j} + w_{j,q}(d_{M_j,M_q} − d_{M_i,M_q}) ;

DELETE(H, i)
    for each unlocked region p ∈ Adj[i] do
        delete swap (i, p) from H ;
```

Figure 4.12: KL algorithm to carry out one-to-one mapping.

Within the inside loop, the gains of $P/2$ swaps with decreasing magnitude order are computed as if the swaps were realized. Here, P is the number of rectangles and also the number of processors. Thus, there are totally $P/2$ possible swaps. Choosing a swap with a maximum gain among the remaining possible swaps will cause a few updates: first, the rectangles and processors involved in the swap can no longer be swapped, thus removed from U containing the unlocked rectangles and the heap H of gains. The gains computed in $INIT\_GAIN$ are updated, since the rectangles are on new processors and their adjacent regions are now farther or nearer to them. $Adj[i]$ yields the adjacent rectangles of rectangle i.

4.6 The Results

Determining the criteria for evaluation of parallel rendering algorithms is crucial to achieve an efficient parallel algorithm. This will enable us to set up the major targets to be attacked while developing efficient parallel rendering algorithms. In general, a parallel rendering algorithm is analyzed in terms of the following issues [33]:

1. Load balancing
2. Data access and distribution
3. Usage of graphical coherence
4. Nature of parallelism
5. Scalability
6. Level of granularity

Although all above factors have more or less the same degree of importance depending on the rendering method and the parallel architecture used, in this section, we have evaluated the BBSP method in terms of the first two issues; namely, the computational

load balance of tasks and the distribution of the object space data. Comments on other issues will also be given throughout the next section. Note that these factors are related to each other in some ways. For example, the data access and distribution might improve the load balance. In the following sections, the two factors that are considered are discussed and comparisons are presented.

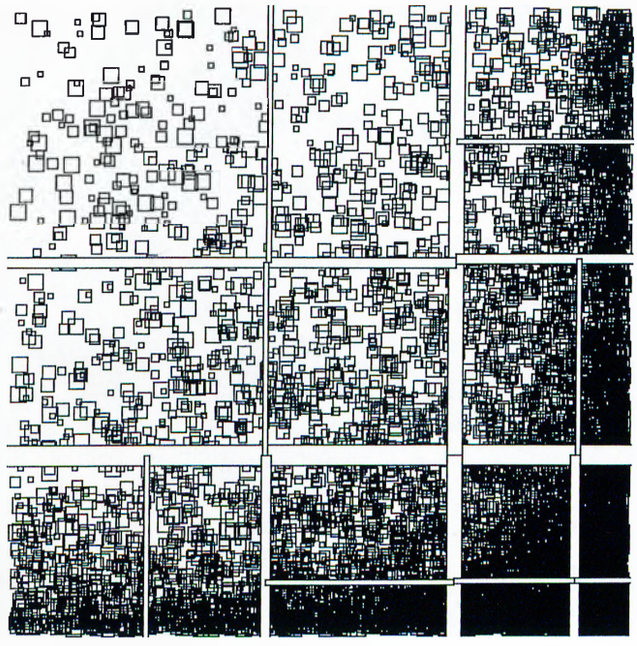
4.6.1 Load Balancing

The most important issue of parallel programs is to be able to maintain an even work load among processors. The finishing time of a processor should be almost the same as the finishing time of other processors. Otherwise, the processors that finish their jobs earlier than others will be idle until the executing processors are terminated. The termination time of the whole program is thus equal to the last termination time of the processors. This will result in the inefficient utilization of processors. The exact measurement of the load imbalance is too difficult to formulate since many factors contribute to the overall load imbalance overhead. But to measure the load imbalance roughly, the first and the average finishing times should be taken into account as in Equation 4.8:

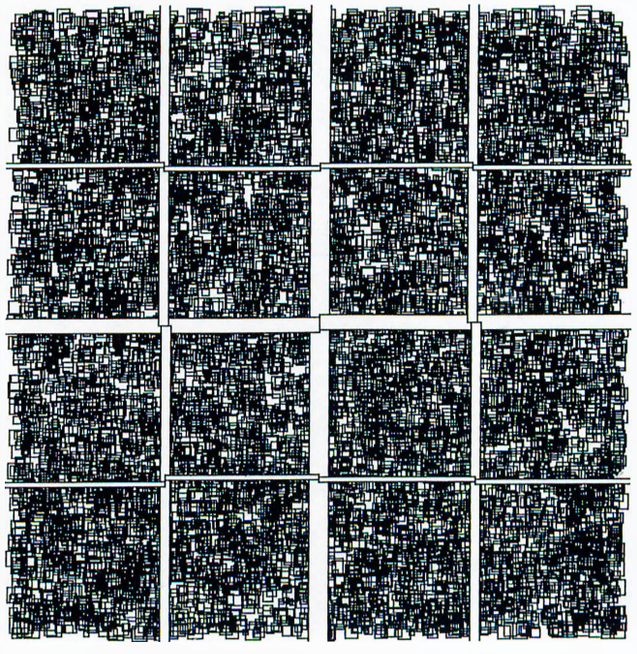
$$Load\ Imbalance = \frac{T_{max} - T_{avg}}{T_{avg}} \tag{4.8}$$

Here T_{avg} and T_{max} are the average and the last finishing times, respectively, of processors executing the tasks of a parallel program. T_{avg} is computed as $\sum_{i=1}^{P} T_i/P$, where P is the number of processors and T_i is the finishing time of i^{th} processor. The Equation 4.8 yields the ratio of the load imbalance overhead for a parallel program. In an ideal case T_{max} and T_{avg} are equal to each other which means that load imbalance ratio is 0. The maximum load imbalance ratio is reached when $T_{avg} = T_{max}/P$. The above equation does not indeed reflect the overall program situation in terms of load imbalance. Because here the other factors (overheads) affecting the termination times are not considered at all. Communication overhead, object access and distribution are some of these factors.

Figures 4.15, 4.16, 4.17 and 4.18 denote the computational and storage imbalance

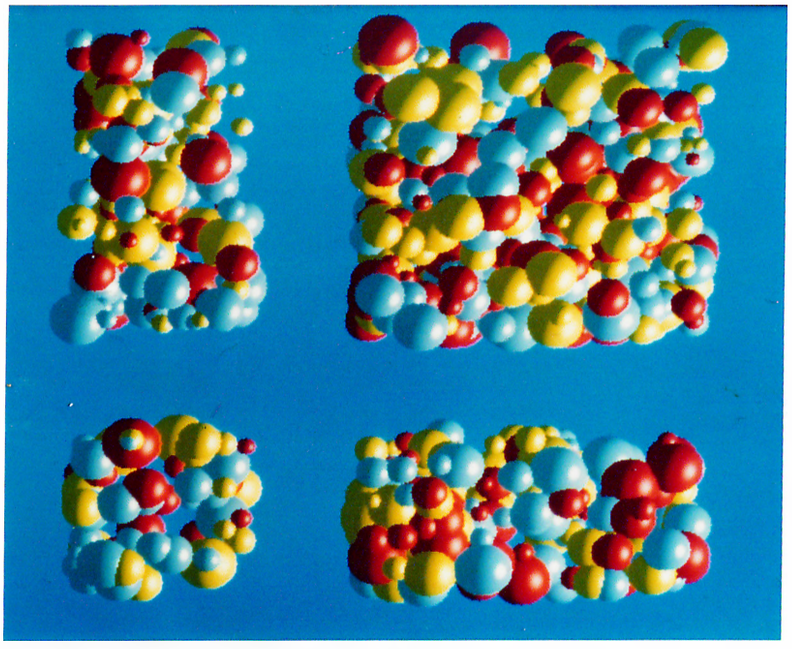


(a)

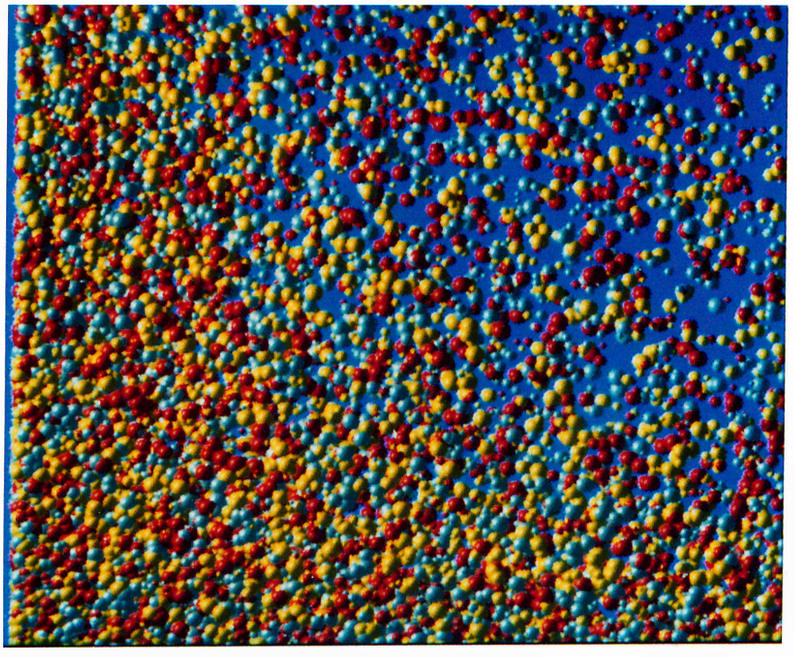


(b)

Figure 4.13: Two types of scenes with N = 10K objects, (a) Gamma (b) Uniform distribution and their subdivision to 16 processors



(a)



(b)

Figure 4.14: Two ray-casted images containing (a) N = 1K (b) N = 30K objects distributed with Gamma probability function.

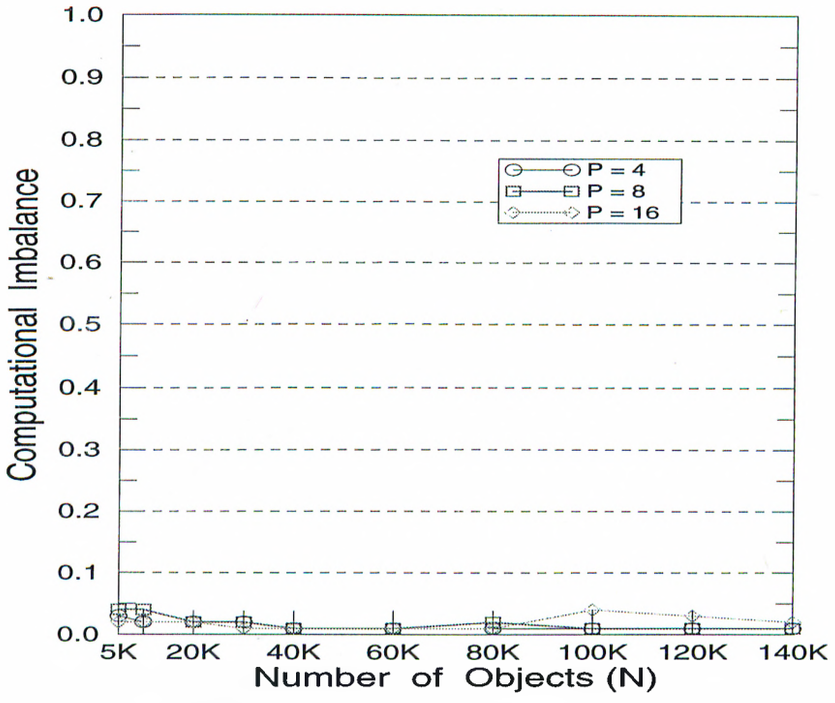


Figure 4.15: Computational imbalance with respect to the total number of objects in a Uniform scene

of two scenes shown in Figure 4.13 in 2-D where the splitting planes are shown by long rectangles with different widths representing the level of the subdivision. Figure 4.14 contains two ray-casted images produced using the developed decomposition and mapping algorithms. In a Uniform scene, the objects are distributed using the Uniform probability distribution function. In a Gamma scene, the objects are distributed according to the Gamma probability distribution function. The Gamma distribution provides a cluster of objects. Since BBSP is based on static load balancing strategy, we give only computational load imbalance figures. BBSP does not need communication between processors, because ray casting is used and each processor stores all of the objects required for intersection tests. The computational load balance is maintained for a Uniform scene. However, the computational load imbalance figures are not good for Gamma scene as seen Figure 4.16. This is actually not expected. We can understand the reason of this by looking at the corresponding storage imbalance figures (Figure 4.18). We have then discovered that the storage imbalance is caused by the shared objects generated as a result of spatial subdivision. We tried to solve this problem using a new

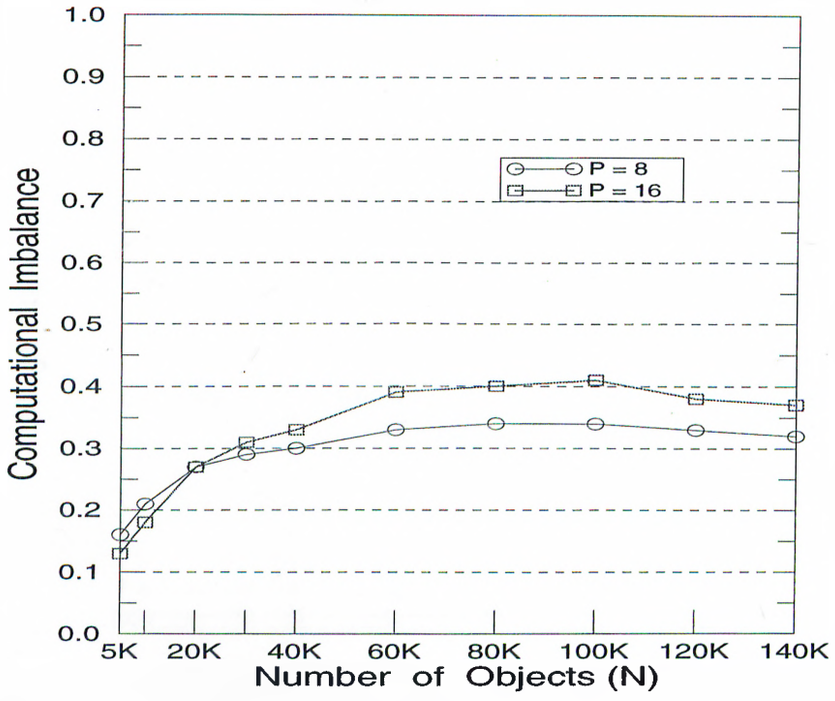


Figure 4.16: Computational imbalance with respect to the total number of objects in a Gamma scene

splitting plane as discussed in Chapter 6.

4.6.2 Data Access and Distribution

In a parallel rendering algorithm using the data parallelization scheme, the scene database is decomposed into several disjoint parts which are then stored in the local memories of the processors in order to handle large scene databases. During the execution of the algorithm, the processors might need to access other parts of the scene database that do not exist in their memory module. The cost of accessing the other memory modules can vary depending on the interconnection topology of the parallel computer and the locations of the communicating processors within the topology graph. The cost of the intercommunication between the processors directly influences the performance of the parallel program. Therefore the assignment of both computational tasks and the parts of the scene database to the processors should be carried out so that the intercommunication between processors is minimized. The BBSP scheme tries to store the close objects in

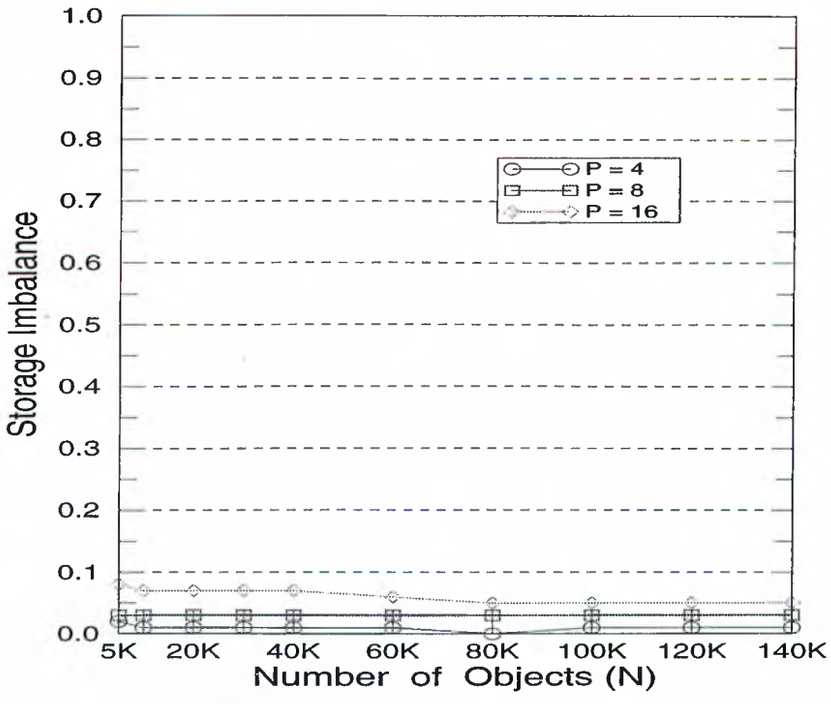


Figure 4.17: Storage imbalance with respect to the total number of objects in a Uniform scene

the local memory of close processors in order to minimize the communication overhead. To achieve this, the neighboring regions resulted from the subdivision are mapped to the neighboring processors.

KL algorithm is used to evaluate data access and distribution of BBSP. KL is initialized in two different ways: random mapping and BBSP mapping. As seen in Table 4.1, the cost of the BBSP mapping is close to the resultant cost of the KL algorithm for most runs, although KL requires more time than BBSP mapping which is performed during subdivision.

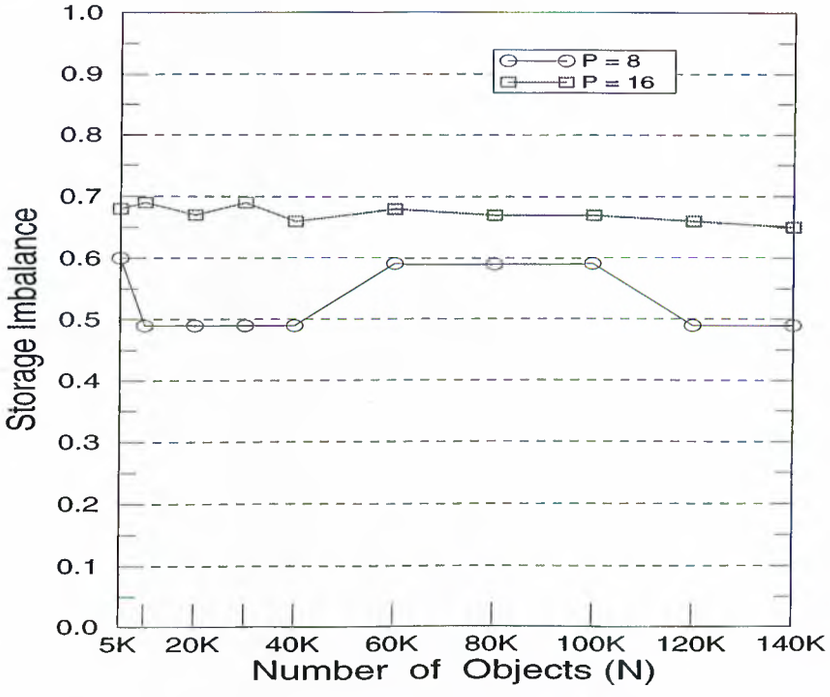


Figure 4.18: Storage imbalance with respect to the total number of objects in a Gamma scene

number of objects	*P*	*initial mapping*					
		random			*gray code ordering*		
		initial cost	*no. of passes*	*final cost*	*initial cost*	*no. of passes*	*final cost*
10K	8	3749	1	2220	2486	1	2220
	16	7108	5	3798	4137	1	3890
	32	14192	4	6952	7284	1	6594
	64	22927	7	11150	10956	3	9406
	128	39845	8	17466	17463	5	15466
	256	64593	7	26774	26737	3	22782
	512	104067	15	38592	39924	5	33747
30K	8	3535	2	2091	2254	1	2091
	16	7369	5	3852	4048	1	3661
	32	14080	3	6847	6723	1	6366
	64	21552	6	10688	10725	3	8806
	128	40479	7	16853	18002	7	14928
	256	63697	11	25168	26357	6	21482
	512	105429	13	39296	41436	5	33903
50K	8	2589	1	2358	2165	1	2090
	16	6670	2	4217	4033	1	3685
	32	12868	8	6682	7010	2	6485
	64	23734	4	10953	10821	3	8806
	128	38433	10	16655	17890	3	14722
	256	64994	7	26535	25736	7	21393
	512	101898	13	38265	38296	4	32219
70K	8	3134	2	2093	2093	0	2093
	16	7551	2	3817	3875	1	3627
	32	13614	6	6549	6845	2	6331
	64	21331	7	10006	10760	3	8936
	128	40278	5	17535	17968	4	14747
	256	64742	14	25055	25993	5	21773
	512	100764	10	38422	38276	6	30800
90K	8	3871	2	2091	2091	0	2091
	16	6885	2	3667	3878	1	3628
	32	13806	4	6597	6844	3	6301
	64	21692	9	9838	10756	3	8803
	128	39682	7	16714	18046	4	15072
	256	65273	11	25059	26035	6	21935
	512	103329	11	38478	38312	6	31217

Table 4.1: Results for scenes with different number of objects.

Chapter 5

Parallel Spatial Subdivision

The spatial subdivision problem is a preprocessing overhead introduced for the efficient implementation of the object-based parallel ray tracing on the target multicomputer. If the spatial subdivision algorithm is implemented sequentially, this preprocessing can be considered in the serial portion of the parallel ray tracing which limits the maximum parallel efficiency. For a fixed input scene instance, the execution times of the parallel ray tracing and the sequential subdivision programs are expected to decrease and increase, respectively, with increasing number of processors in the target multicomputer. Thus, this preprocessing will begin to constitute a drastic limit on the maximum efficiency of the overall parallelization due to Amdahl's law [28]. Hence, parallelization of the subdivision algorithm on the target multicomputer is a crucial issue for efficient object-based parallel ray tracing. In this chapter, we propose an efficient parallel spatial subdivision algorithm to utilize the processors of the target multicomputer to be used for object-based parallel ray tracing algorithm. After an initial random distribution of objects to processors, objects intermittently migrate during the execution of the recursive bisection algorithm in accordance with the mapping strategy such that all objects arrive at their *home* processors at the end of the parallel subdivision process. Each object traverses at most $\log_2 P$ processors to reach its home processor.

5.1 A Parallel Spatial Subdivision Algorithm

For complex scenes, spatial subdivision using the proposed BBSP scheme still may take too much time. For that reason, we can use the node processors of the target multicomputer to speed up the subdivision process. Furthermore, these processors are already idle waiting for the start of the ray-tracing-loop. This approach increases the utilization of the parallel system. Reducing the spatial subdivision time is also studied by other researchers. McNeill et al. [22] have suggested an algorithm for dynamic building of the octree to reduce the data structure generation time. In this section, we propose a parallel subdivision algorithm - a parallel version of BBSP scheme for hypercube multicomputers. The proposed BBSP algorithm is based on divide-and-conquer paradigm. Hence, BBSP algorithm is very suitable for parallelization on hypercubes due to their recursive structures mentioned earlier. The proposed parallel BBSP algorithm has a very regular communication structure and requires only concurrent single-hop communications (i.e., communications between neighbor processors) on hypercubes. The proposed parallel BBSP algorithm may also be adopted to other interconnection topologies. However, multi-hop communications may be required in other topologies.

In the proposed scheme, host processor randomly decomposes the object database into P even subsets such that each subset contains either $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$ objects and it sends each subset to a different node processor of the hypercube. Then, the following steps are performed in a divide-and-conquer manner ($d = \log_2 P$ times) for each channel c from $c = d-1$ down to $c = 0$.

Step 1 Node processors concurrently construct their local integer arrays corresponding to their local object database.

Step 2 Processors concurrently perform *prefix-sum* operation on their local integer arrays.

Step 3 Processors of each $(c+1)$-dimensional disjoint subcube perform global *vector*

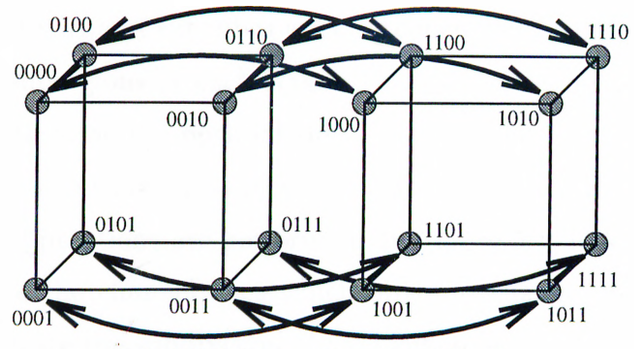
sum operation on their local integer arrays. Note that there exist 2^{d-c-1} global vector-sum operations performed concurrently. At the end of this step, processors of each $(c+1)$-dimensional subcube will accumulate the same local copies of the prefix-summed integer-arrays.

Step 4 Replicated integer arrays on x and y dimensions in each subcube are virtually divided into 2^{c+1} even slices and each slice is assigned to a different processor of that subcube. Then, processors perform the cost computations of the splitting planes corresponding to their slices in order to find their local optimal splitting planes.

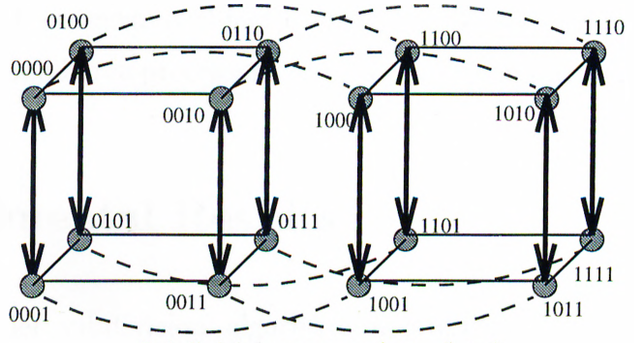
Step 5 Processors of each subcube perform a global *minimum* operation to locate the optimal splitting plane corresponding to the subregion mapped to that subcube.

Step 6 Processors of each subcube determine their local subsubregion assignment, for the following stage $c-1$, according to the proposed mapping scheme. Then, processors concurrently perform a single pass over their local object database to gather and send the objects which belong to the other subsubregion to their neighbors on channel c. Hence, two subsubcubes of each subcube effectively exchange their subset of local object databases such that each subsubcube collects the object database corresponding to their subsubregion assignment in the following stage $c-1$. Note that 2^{d-c-1} subsubcube pairs perform such exchange operation concurrently.

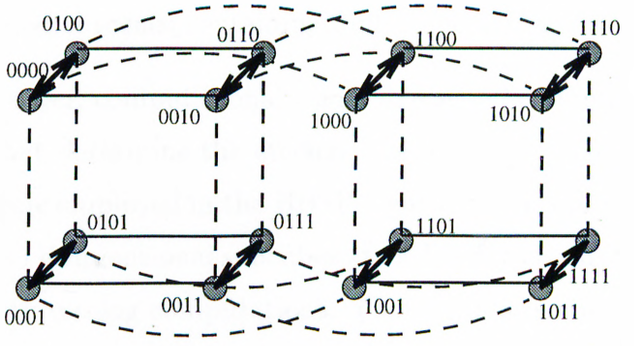
During Step 6, processor pairs also determine their local shared objects which are not involved in the exchange operation. However, processors update either $xmin$ ($ymin$) or $xmax$ ($ymax$) values of their local shared objects according to their subsubregion assignment for a vertical (horizontal) splitting plane. Hence, processors maintain and process disjoint rectangular parts of the bounding boxes corresponding to the shared objects. Figure 5.1 illustrates the operation structure of the proposed parallel BBSP algorithm on 4-dimensional hypercube topology. In this figure, links drawn as dashed lines illustrate the idle links in a particular stage of the parallel algorithm. Links drawn as



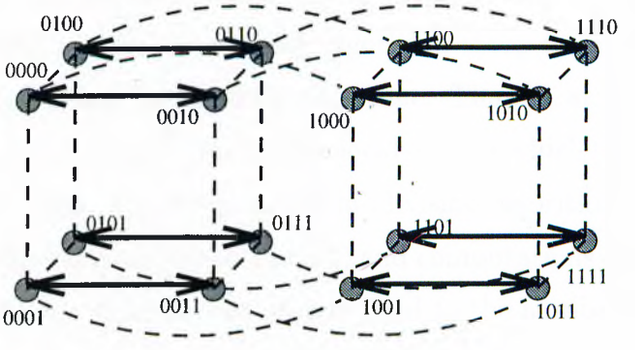
Subdivision accross channel c=3



Subdivision accross channel c=2



Subdivision accross channel c=1



Subdivision accross channel c=0

Figure 5.1: Tearing of hypercube topology as the subdivision proceeds.

solid lines illustrate the disjoint subcubes working concurrently and independently for the subdivision of their subregions at each stage. That is, processors of each subcube work in cooperation to determine the optimal subdivision of the subregion assigned to that subcube. These links also show the subcubes in which intra-subcube global vector-sum and global minimum operations are performed. In Figure 5.1, links drawn as solid lines with arrows illustrate the channel over which object-exchange operation takes places. These links also illustrate the subdivision of each subcube into two disjoint subsubcubes at the end of each stage. As is also seen in Figure 5.1, all objects arrive at their home processors after $\log_2 P$ concurrent object-exchange operations. Note that shared objects will have more than one home processors and they will be replicated in those processors.

5.2 Experimental Results

The proposed parallel subdivision algorithm is implemented on an Intel's iPSC/2 hypercube multicomputer with 16 processors. The performance of the parallel program is experimented on several scenes containing different number of objects.

As is mentioned earlier, computational load balance and communication overhead are two crucial factors that determine the efficiency of a parallel algorithm. The recursive spatial bisection scheme employed in the BBSP algorithm tries to maintain load balance among the disjoint $(c+1)$-dimensional subcubes at each subdivision stage c during the first level of the parallel ray tracing computations. That is, in a particular subdivision stage c, the products of the number of local objects and areas of the rectangular subregions assigned to disjoint subcubes are approximately equal to each other. Note that, at the end of each stage of the parallel subdivision algorithm (Step 6), objects always migrate to their destination subcubes for the following stage. That is, at the beginning of each subdivision stage, each subcube holds only the local objects which belong to its respective local rectangular subregion. However, in the subdivision algorithm, the complexities of local object-based computations (Steps 1 and 6) and computations on local integer arrays (Steps 2, 3 and 4) within a subcube are proportional to the number of local objects and

the semi-parameter (height+width), respectively, of the rectangular subregion assigned to that subcube. Hence, the complexities of local computations within a subcube during the parallel ray tracing and the parallel subdivision algorithms depend on the same factors; number of local objects, height and width of the rectangular subregion assigned to that subcube. However, the dependence is multiplicative in the parallel ray tracing, whereas, it is additive in the parallel subdivision. Hence, this deviation in the load balance measures of these two parallel algorithms may introduce load imbalance among subcubes during the parallel subdivision since the proposed parallel subdivision algorithm inherently operates in accordance with the mapping strategy adopted by the recursive spatial bisection scheme which tries to maintain a load balance during parallel ray tracing. This type of load imbalance is referred here as *inter-subcube* imbalance. There exists no load imbalance among the processors of the individual subcubes during the local integer computations at Steps 2, 3 and 4, since each processor of a subcube operates on local integer arrays of the same size. However, processors of the same subcube may hold different number of local objects belonging to the respective subregion during a particular stage of the algorithm. This type of load imbalance, which is referred here as *intra-subcube* imbalance, may introduce imbalance during the concurrent object-based computations (Steps 1 and 6) between the processors of the same subcube. Intra-subcube load imbalance may introduce processor idle time both during the global synchronization at Step 3 (global vector-sum operation) and object exchange synchronization at Step 6 within subcubes. Initial *random* distribution of objects to processors is an attempt to reduce intra-subcube load imbalances.

The communication overhead of the proposed parallel algorithm involves two components; number and volume of communication. In a medium-to-coarse grain architecture with high communication latency, the number of communications may be a crucial factor affecting the performance of the parallel algorithm. Each one of the intra-subcube global operations at Steps 3 and 5 require $c+1$ concurrent exchange communication steps at stage c. Under perfect load balance conditions, these global communications within different subcubes will be performed concurrently. Hence, the total number of concurrent communications due to these intra-subcube global operations

is $d(d+1)$. Thus, the total number of concurrent communications becomes $d(d+2)$ since the object exchange operations (Step 6) require d concurrent communications in total under perfect load balance conditions. Hence, percent overhead due to the number of communications is negligible for sufficiently large granularity (N/P) values.

The volume of concurrent communication during an individual intra-subcube global minimum operation (Step 5) is only $2(c+1)$ integers at stage c. On the other hand, the volume of the concurrent communication during an individual intra-subcube global vector-sum operation (Step 3) is $2(c+1)(n+m)$ integers where $n+m$ denotes the semi-perimeter of the rectangular subregion assigned to that subcube at stage c. That is, the total volume of this type of communications depend on the semi-perimeter of the initial window and d. Hence, percent overhead due to these types of integer communications decreases with increasing scene complexity for a fixed window size. The total volume of communication due to the object migrations is a more crucial factor in the parallel performance of the proposed parallel algorithm. Under average-case conditions, half of the objects can be assumed to migrate at each stage of the algorithm. Hence, if shared objects are ignored, the total volume of communications due to object migrations can be assumed to be $(N/2)\log_2 P$ objects. Experiments on various scenes yield results very close to this average-case behavior.

Under perfect load balance conditions, each processor is expected to hold N/P objects and each processor pair can be assumed to exchange $N/2P$ objects, at each stage. Hence, under these conditions total concurrent volume of communications due to object migrations will be $(N/2P)\log_2 P$ objects. Experiments on various Uniform scenes yield results very close to these expectations. However, results slightly deviate from these expectations for non-uniform (Gamma) scenes with objects clustered toward particular positions.

Figure 5.2 illustrates the efficiency curves for different dimensional hypercubes as function of the scene complexity. Efficiency values on a hypercube with P processors are computed as $E_p = T_1/PT_p$ where T_1 and T_p denote the execution times of the sequential and parallel subdivision programs on 1 and P node processors, respectively. As is seen

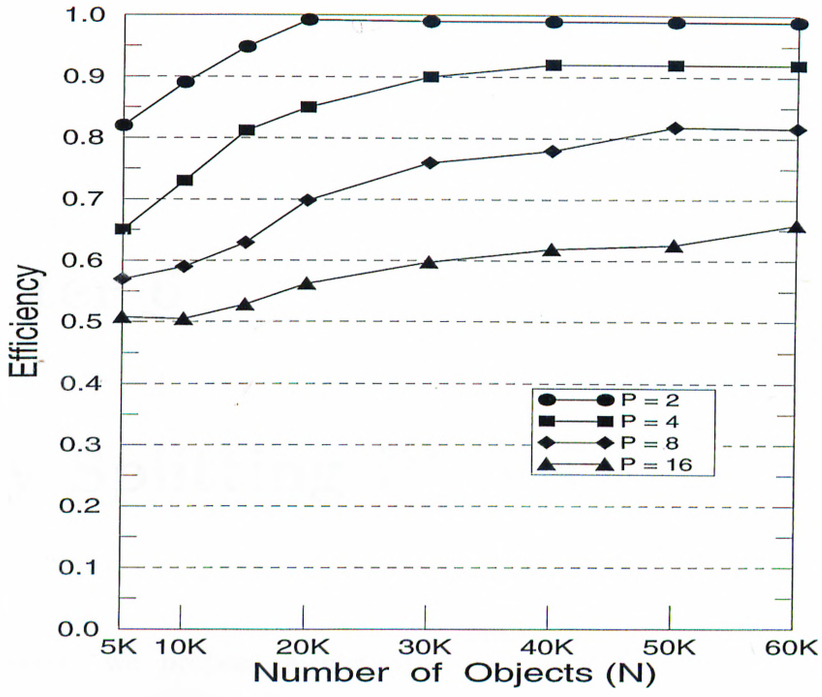


Figure 5.2: Efficiency curves with respect to the total number of objects in the scene

in Figure 5.2, efficiency increases with increasing scene complexity and fixed window resolution size. This increase can be attributed to two factors. The total number of communications stays fixed for a fixed hypercube size. Hence, percent overhead due to the total number of communications decreases with increasing scene complexity. Similarly, the volume of integer communications also stays fixed for fixed hypercube size and window resolution size. Hence, percent overhead due to the volume of integer communications also decrease with increasing scene complexity. As is seen in Figure 5.2, efficiency values close to 100% are obtained for $P = 2$ processors since initial even distribution of objects entirely avoids both intra- and inter-subcube load imbalances during the first stage of the parallel BBSP algorithm. However, for a fixed scene instance, efficiency decreases considerably with increasing number of processors. This decrease is mainly due to the increase in the inter-subcube load imbalances, since each doubling of the number of processors introduces an extra stage to the algorithm. Therefore, load re-balancing algorithms should be developed for larger number of processors.

Chapter 6

Jaggy Splitting Planes

In this chapter, we propose a new splitting plane concept the so called *jaggy splitting plane* to accomplish full utilization of memory space of processor nodes of a multicomputer. Jaggy splitting planes also eliminate duplicate computations due to duplicated objects stored in the local memory of processors. Jaggy splitting planes accomplish these by avoiding the shared objects which are the major sources of inefficiencies for both memory and processor utilizations. The gains obtained are shown on sample scenes in the following sections.

6.1 Side Effects of Shared Objects

We prefer spatial subdivision method in sequential and parallel acceleration approach, since it allows us to exploit spatial coherence property. Another advantage of spatial subdivision is the generation of disjoint volumes that decrease the duplicate computations for intersection tests. Additionally, when used in a parallel acceleration approach, spatial subdivision provides data coherence that minimizes the communication among processors. Bounding volumes method which is a major alternative to spatial subdivision can not provide spatial and data coherence easily. In other words, to construct a good

hierarchy of bounding volumes that tightly surround the objects is in general too hard, even if user intervention is enabled.

On the other hand, the spatial subdivision suffers from the shared objects. The problems caused by the shared objects were discussed in Chapter 4: First they are duplicated in the local memory of processors that are assigned 3-D subvolumes intersecting the shared objects. Second, an intersection test with a shared object is repeated if the computed intersection point is not inside the subvolume of the processor performing the test. Because, there can be another closer intersection point within the next subvolume along the path of the ray. Finally, shared objects increase the number of objects that are taken into account for the intersection tests performed by the processors of the multicomputer system. That is, duplication of shared objects not only waste memory space but also result in the duplicate computations for intersection tests as discussed in the following sections.

6.1.1 Wasting Memory

Let us consider a scene with N objects that will be distributed to the processors P_i, where $i = 1..P$ and P is the number of processors in the multicomputer. Ideally, each processor P_i is supposed to store and handle N/P objects in the subvolume assigned to them. However, the number of objects stored in the local memory of each processor is greater than N/P due to the shared objects and varies between processors depending on the spatial distribution of objects in the scene. If N_i is the number of objects and S_i is the number of shared objects in the processor P_i, then the total number of objects in the processor P_i will become $N_i = N/P + S_i$. The total amount of memory waste is $\sum_{i=1}^{P} S_i$. Note that the first (original) copy of a shared object is not included in S_i.

The objective function of BBSP used to split a given region contains terms to maintain computational load balance by equating the product of objects and pixels to be assigned to each processor. However, shared objects damage this equality as subdivision proceeds. At the end of subdivision, each processor may have quite different number of objects

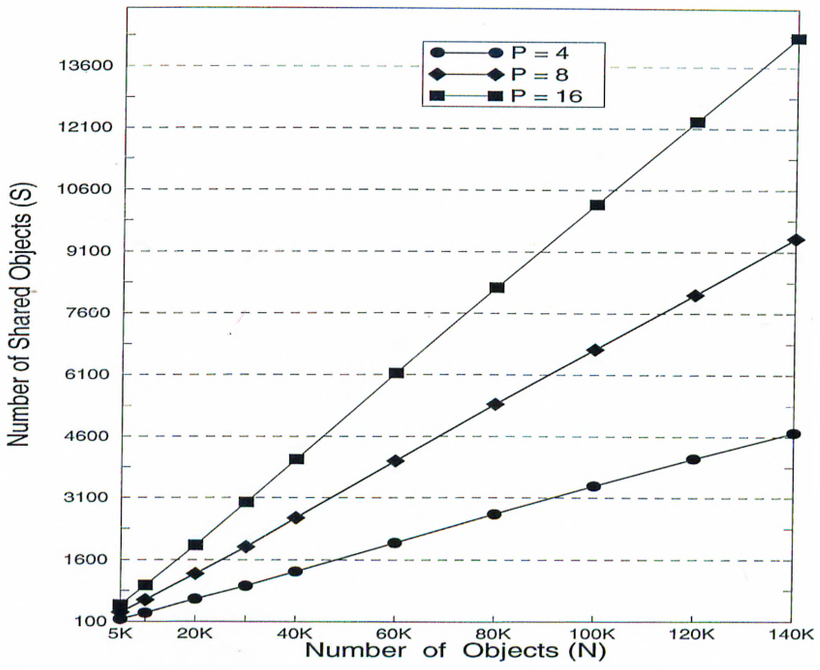


Figure 6.1: Total number of shared objects resulting from uniform scenes with different number of objects

assigned to them. The higher depth of subdivision the more shared objects in the system. For instance, when the depth of subdivision is 6 ($P = 64$), the total number of shared objects may reach half of the objects in a complex scene. This substantially reduces the maximum number of objects in a scene for rendering which contradicts with the major purpose of exploiting parallelism on distributed-memory MIMD which is the huge memory space provided in order to handle large scenes that can not fit into the local memory of each processor. The memory is not wasted due to the shared objects when spatial subdivision or BSP is used in sequential acceleration, since the objects are stored once and only pointers to them are kept in the auxiliary data structure. The number of shared objects generated for various sample scenes and for different subdivision depths are shown in Figure 6.1 and 6.2. These graphs show that the parallel and sequential speed-up values can considerably be affected by the shared objects as will be discussed in the following section. Additionally, the maximum number of objects that can be rendered in a scene is restricted due to inefficient utilization of memory. Since jaggy splitting planes do not allow shared objects, the memory space is used efficiently.

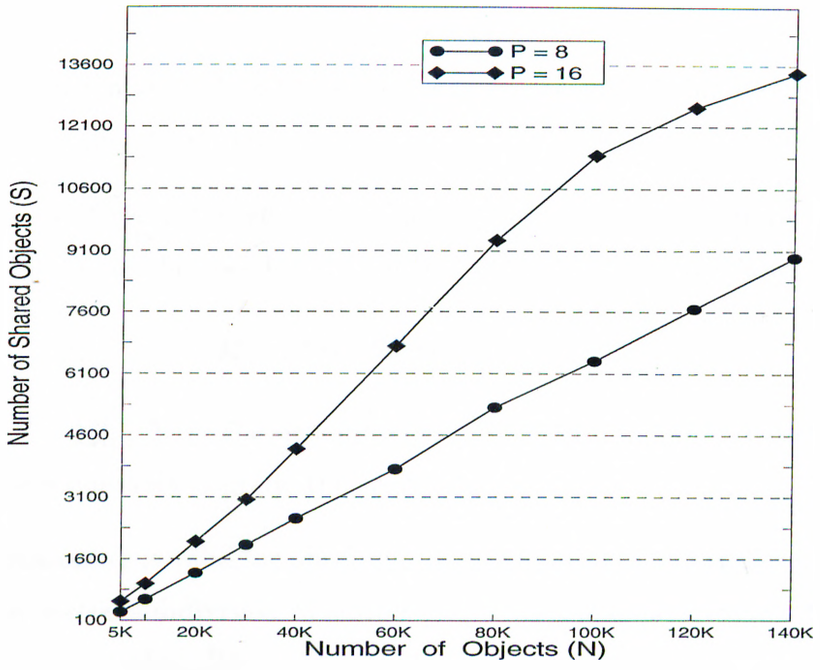


Figure 6.2: Total number of shared objects resulting from Gamma scenes with different number of objects

6.1.2 Duplicate Computations

Shared objects also cause duplicate computations for intersection tests in both sequential and parallel acceleration. Let us consider sequential ray casting of a scene containing N objects. Assume that the depth of BSP subdivision is $log_2 P$, the resolution of the image to be computed is $n \times n$, the number of shared objects is S.

The speed-up obtained can be calculated as the proportion of the time consumed by the naive algorithm T_{naive} to the time consumed by the accelerated one T_{bsp}:

$$Sequential\ Speedup = \frac{n \times n \times N \times O_{test}}{\sum_{i=1}^{P} R_i \times N_i \times O_{test}} \tag{6.1}$$

where O_{test} is the cost of intersection test, P is the number of subvolumes generated, R_i is the number of pixels, and N_i is the number of objects in region i. The speed-up is directly proportional to the number of regions generated P. The maximum speed-up is obtained under the conditions when there is no shared object ($S = 0$ and $N = \sum_{i=1}^{P} N_i$) and the number of pixels R_i and the number of objects N_i associated with each region

is the same $R_i = (n \times n)/P$ and $N_i = N/P$. Therefore, the maximum speed-up is P as long as the mentioned conditions are satisfied. As the subdivision depth increases, these conditions are difficult to satisfy.

Now, let us observe the effect of shared objects to the maximum speed-up value. After simplifications, the speed-up expression can be written as follows:

$$Max.\ Seq.\ Speedup = \frac{P \times N}{N + S} \tag{6.2}$$

where $N + S = \sum_{i=1}^{P} N_i$ in the denominator. As seen in the above equation, the shared objects can considerably degrade the speed-up.

Similar reasoning is valid for obtaining the speed-up of the parallel ray casting using BBSP as the spatial subdivision. The speed-up for parallel acceleration for P processors (subdivision depth = $\log_2 P$) can be stated as the proportion of the best sequential time and the parallel time:

$$Parallel\ Speedup = \frac{(n \times n \times N \times O_{test})/P}{R_i \times N_i \times O_{test}} \tag{6.3}$$

where N_i is the number of objects assigned to the processor P_i and can be obtained as $N_i = N/P + S_i$. P_i is the processor that is the most heavily loaded and thus finishes its task latest. The maximum speed-up can be obtained when the number of objects and the number of pixels are distributed and assigned fairly to the processors ($R_i = (n \times n)/P$ and $N_i = N/P$) that maintains computational load balance among the processors. The maximum speed-up obtained will be P which is linear speed-up.

The effect of shared objects to the maximum speed-up of parallel ray casting can be formulated after simplifications as:

$$Max.\ Par.\ Speedup = \frac{P \times N}{N + P \times S_i} \tag{6.4}$$

As seen in the above equation, the speed-up is drastically degraded by the shared objects. Note that this result is worse than that of sequential, since the run time of the parallel ray casting is determined by the most heavily loaded processor which has probably the greatest number of shared objects.

Having demonstrated the negative effects of shared objects on both sequential and parallel ray casting, we propose a splitting plane concept called as the jaggy splitting plane that will avoid the shared objects in the system. Jaggy splitting planes are described and implemented for parallel ray casting that incorporates the BBSP method for decomposition and mapping of the object-space to the processors. Its adaptation to the sequential algorithm is quite straightforward.

6.2 Modified BBSP

Spatial subdivision method BBSP proposed previously in this research is modified to include jaggy splitting planes. The 3-D space is subdivided into two disjoint subvolumes at each step of the BBSP. The positions of the splitting planes are identified using the same objective function. The non-shared objects in the resultant subvolumes are mapped as in BBSP. The first modification is in the assignment of the shared objects straddling the splitting planes. In the BBSP with conventional splitting planes, the shared objects were duplicated in all subvolumes intersecting them. The second and last modification is in the shape and processing of the pixel regions assigned to the processors. These two modifications are discussed in the following sections.

6.2.1 Assignment of Objects

The only way of avoiding the mentioned side effects of shared objects is not to duplicate them in the local memories of processors. Because when they are duplicated, they both waste memory space and cause the processors to consider them in the intersection tests that results in the enlargement of the search space to find the first intersection point.

The 3-D scene is subdivided into rectangular prisms which are mapped to the processors as before. The position of a splitting plane is found out using the same objective function. When a splitting plane is identified, the objects completely on the left and on the right of the splitting plane are assigned to the corresponding regions.

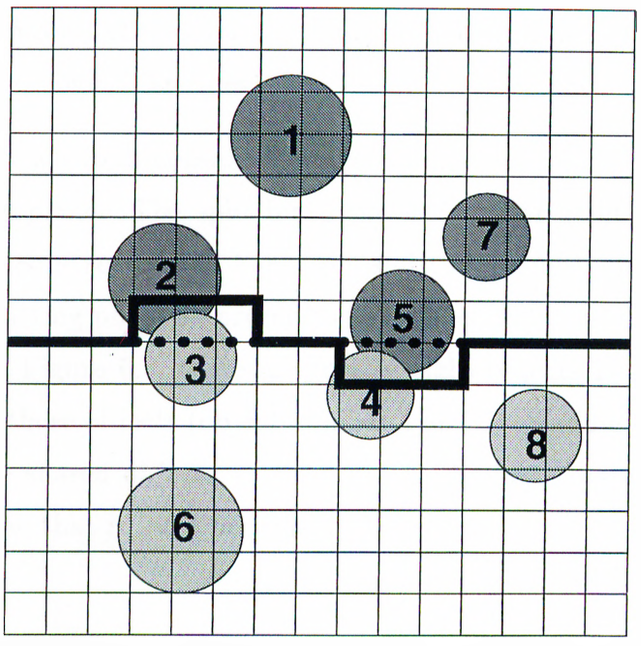


Figure 6.3: Two-dimensional representation of a jaggy splitting plane consists of a set of line segments, not a single straight line.

However, the shared objects that straddle the splitting plane are assigned to the left (below) or to the right (above) region on the basis of one of the following conditions which are checked in order of decreasing importance.

1. A shared object is assigned to the region left (below) or right (above) that is most occupied by the shared object in terms of 3-D volume.

2. If the shared object occupies both regions almost in equal proportions, then the object is assigned to the region that, at that instance, holds less number of objects in order to equate the number of objects on both regions.

3. Otherwise, the shared object is randomly assigned to one of the regions.

Each shared object is tested with these conditions in order and it is assigned to one of the regions accordingly. The modified BBSP does not allow any objects to be duplicated. The reasons why these conditions are needed and ordered in this way are to make the

pixel region computations efficient and minimize the communications among processors. These reasons will be obvious and described in the next section.

Figure 6.3 shows a two-dimensional representation of a scene subdivided into two by a jaggy splitting plane. The splitting plane is not a straight line any more, it is composed of a set of line segments. The jaggy splitting plane is obtained by modifying the conventional splitting plane according to the associated shared objects. As seen in the sample scene in Figure 6.3, there are totally eight objects inside the region to be subdivided. Two of these objects (object number 3 and 5) straddle across the computed splitting plane. The shared objects 3 and 5 satisfy the first condition and each one is assigned to the region that is most occupied.

6.2.2 Computing the Pixels

The pixel regions assigned to the processors are rectangular as before. However, every processor is also responsible from the pixels of the extensions due to the shared objects stored in its local memory. The pixels to be computed can belong to three distinct types of pixel regions which are handled differently: *Local, Inside Extension,* and *Outside Extension* as shown in Figure 6.4. Local pixel region of a processor can be computed using the objects in its local memory. Inside and outside extension regions are computed using the objects stored in the local memories of more than one processor. These extension regions contain objects overlapped with respect to the viewing direction and stored in more than one processor. Therefore, the processors containing such overlapping objects find out the values of the pixels in the extensions using their local overlapping objects. These pixel values computed by different processors are then merged using the associated depths with respect to the viewing direction. Inside extension of a processor corresponds to the extension which is included in its rectangular region. Outside extension of a processor is composed of extensions that does not overlap with its rectangular region. The boundaries of these regions are specified by the bounding boxes of the shared objects. Some outside extensions of some processors assigned adjacent

rectangular regions constitute the inside extension of a processor. Such processors are called communicating processors which are connected to each other in the rectangle adjacency graph. Each processor carries out the following steps to compute the values of the pixels it is responsible from:

Step 1 Determine and send the boundaries of the outside extensions using the shared objects to let the communicating processors know and form the corresponding inside extensions.

Step 2 Compute the local pixels and store the depths of the pixels associated with the inside extensions formed from the boundaries sent by the communicating processors.

Step 3 Compute and send the pixel values with their associated depths of the outside extensions to the communicating processors that own the corresponding inside extensions.

Step 4 Merge the inside extension pixels with the corresponding outside extension pixels received from the communicating processors.

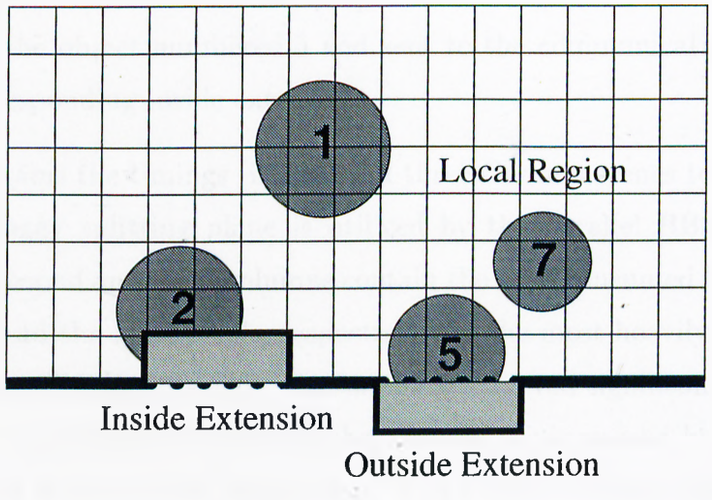


Figure 6.4: Local, inside extension and outside extension of a sample scene.

Since the communicating processors that are mapped adjacent regions will need to communicate, the mapping should be done in such a way that the required

number of objects	*jaggy*		*conventional*
	extension	*local*	*total*
512	42	41150	49002
10240	2472	392679	634327
30720	9860	1672119	2710401

Table 6.1: Timings in msec for scenes with different number of objects.

communication is minimized. Each processor will communicate with its communicating processors two times at Step 1 and 3. The communication at the first step is synchronous and the its volume is not high since it includes only boundary definitions. The communication at the third step may have high volume of communication, but it is asynchronous and can be performed after computing the local pixel values.

Figure 6.4 contains a rectangular region with outside and inside extensions to be computed by a processor. Note that the inside extension is overlapped by the non-shared local object 2. This processor should store the depths for the inside extension that will be merged with outside extensions computed and sent by the corresponding communicating processors. The pixels of the outside extension shown in Figure 6.4 are computed using the object numbered 5 and sent to the communicating processor that contains the corresponding inside extension.

Table 6.1 contains the timings (in msec) of three Gamma scenes to denote the gains obtained when jaggy splitting plane is utilized by the parallel BBSP scheme on 16 processors. The second and third columns contain the time consumed for computing the extension pixels and the local pixels, respectively, in the most heavily loaded processor that finishes latest. The results show that we have achieved significant gain with jaggy splitting planes. Note that when the number objects are increased by three times, the speed-up obtained is not at that proportion. The reason is that we also use BBSP in each processor sequentially.

Chapter 7

Summary, Contributions and Future Work

7.1 Summary

Computer images generated from mathematical definitions are widely used in many computer graphics applications such as animation, visualization, simulation, education, architecture, advertising and medicine. Ray tracing and ray casting are two well-known rendering methods to generate such images. Although these rendering methods can produce very realistic views to achieve their task, they are too slow for an interactive system especially when complex scenes are to be considered. Exploiting parallelism is essential for interactive display of complex scenes using these rendering algorithms.

This research has concentrated on the parallelization of ray casting/tracing rendering methods on distributed-memory MIMD parallel architectures. The parallel architecture chosen provides not only processing processing power but also large memory space that allows complex scenes to be rendered interactively. In order to render complex scenes on such parallel architectures efficiently, both computations and scene objects need to be decomposed and mapped to the processors so that the interprocessor communication

is minimized and the load balance among processors is maintained. Besides, the same computations and processing should not be carried out by the processors redundantly.

In this dissertation, we have adapted the spatial subdivision method BSP to the parallel ray casting/tracing for accomplishing the decomposition and mapping tasks efficiently. Due to the unprecedented paths of the secondary rays (shadow, reflection etc. rays) that depend on the orientation of the objects in the scene, it is very hard to maintain the load balance for the secondary rays using a static load balancing strategy on which the proposed algorithm is based. The proposed subdivision algorithm maintains the load balance particularly for the primary rays. The algorithm is, therefore, more suitable for ray casting. However, it can also be used for the initial assignment of 3-D regions to the processors for a parallel ray tracing using caching mechanism.

Decomposition and mapping tasks which are performed during the preprocessing can take excessive time for complex scenes. The time overhead caused by these tasks can be reduced by utilizing the processors of the parallel architecture that is waiting idle for executing the rendering code. An efficient parallel BBSP algorithm is developed to decrease the preprocessing time.

Although the spatial subdivision accelerating ray casting/tracing has many advantages over its alternatives, it owns the problem of shared objects which degrade the performance of both sequential and parallel ray casting/tracing algorithms. Jaggy splitting plane is introduced to avoid the mentioned problem of the spatial subdivision.

7.2 Contributions

The salient contributions of this research can be summarized as follows:

- A new method called BBSP based on spatial subdivision is developed to efficiently perform decomposition and mapping tasks for parallel ray casting/tracing. The objective function incorporated by BBSP includes terms to maintain load balance

among processors and minimize interprocessor communication while optimizing the number of shared objects. The positions of splitting planes are identified using only integer additions.

- An algorithm has been developed to find the neighbors of a region in BBSP method. The proposed neighbor-finding algorithm is based on the data structures used in BBSP and can be implemented easily.

- An efficient parallel spatial subdivision algorithm based on BBSP has been proposed to achieve the mapping of objects to processors simultaneously with decomposition of the scene. The parallel algorithm avoids redundancy by parallelizing most of the operations such as establishing the data structures and identifying the optimal splitting planes.

- Jaggy splitting plane is proposed to eliminate the side effects of spatial subdivision due to the shared objects. The side effects include poor utilization of memory space and duplicate intersection tests for shared objects. The parallel ray casting/tracing algorithms have been modified to employ the jaggy splitting plane.

7.3 Further Research Areas

In this research, the considered scene is composed of geometric objects such as spheres, cylinders, cones, superquadrics and polygons. The proposed BBSP method can be adapted to the direct volume rendering used to create an image from volumetric data sets without generating an intermediate geometrical representation [32]. In the simplest case, the volumetric data consists of single scalar values located in 3-D space.

The domain-mapping problem in different rendering methods can be studied to obtain more accurate images at faster rates. For example, the Monte Carlo Ray-tracing and the Radiosity with reflections and refractions can be taken into account for parallelization. The major advantage of Radiosity is that it models global illumination completely. Monte

Carlo Ray-tracing method, on the other hand, tries to calculate the indirect illumination with statistical methods.

The same problem (rendering of complex scenes) can be considered in a distributed environment where a number of workstations are connected to each other with fast communication links. This will be useful for film generation involving rendering of complex scenes. The idle times of the workstations can be utilized for this purpose in batch mode, when parallel architectures are not available.

Bibliography

[1] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, May 1987.

[2] T. Bultan and C. Aykanat. A new mapping heuristic based on mean field annealing. *Journal of Parallel and Distributed Computing*, 16:292–305, 1992.

[3] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Communications of the ACM*, 37(4):31–41, April 1994.

[4] M. B. Carter and K. A. Teague. Distributed object database ray tracing on the intel ipsc/2 hypercube. Technical report, Dept. of Electrical and Computer Eng., Oklahama State University, USA, 1990.

[5] E. Caspary and I. D. Scherson. A self-balanced parallel processing for computer vision and display. In P. M. Dew, T. R. Heywood, and R. A. Earnshaw, editors, *Parallel Processing for Computer Vision and Display*, pages 408–419. Addison-Wesley, January 1989.

[6] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, 5:3–12, 1986.

[7] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computation. *Information Processing Letters*, 16:217–219, 1983.

[8] M. Dippé and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics*, 18(3):149–158, July 1984.

[9] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 5(2):16–26, April 1985.

[10] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[11] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(3):14–20, May 1987.

[12] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, 6:12–26, November 1989.

[13] S. A. Green and D. J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 2(6):62–73, 1990.

[14] S. A. Green, D. J. Paddon, and E. Lewis. A parallel algorithm and tree-based computer architecture for ray-traced computer graphics. In P. M. Dew, T. R. Heywood, and R. A. Earnshaw, editors, *Parallel Processing for Computer Vision and Display*, pages 431–442. Addison-Wesley, January 1989.

[15] V. İşler, C. Aykanat, and B. Özgüç. Subdivision of 3d space based on the graph partitioning for parallel ray tracing. In P. Brunet and F. W. Jansen, editors, *Photorealistic Rendering in Computer Graphics*, pages 182–190. Springer-Verlag, Berlin Heidelberg, Germany, February 1994.

[16] M. R. Kaplan. The use of spatial coherence in ray tracing. In D. E. Rogers and R. A. Earnshaw, editors, *Techniques for Computer Graphics*, pages 173–193. Springer-Verlag, New York, 1987.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[18] H. Kobayashi and T. Nakamura. Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, 3(1):13–22, 1987.

[19] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 2(4):197–209, 1988.

[20] INMOS Ltd. *Occam Programming Manual.* Prentice-Hall, London, 1984.

[21] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 3(6):153–166, 1990.

[22] M. D. J. McNeill, B. C. Shah, M. P. Hébert, P. F. Lister, and R. L. Grimsdale. Performance of space subdivision techniques in ray tracing. *Computer Graphics Forum*, 11(4):213–220, 1992.

[23] K. Nemoto and T. Omachi. An adaptive subdivision by sliding boundary surfaces. In *Proceedings: Graphics and Vision Interface '86*, pages 43–48. Canadian Information Society, Toronto, 1986.

[24] F. Özgüner and C. Aykanat. A reconfiguration algorithm for fault tolerance in a hypercube multiprocessor. *Information Processing Letters*, 29:247–254, 1988.

[25] D. J. Plunkett and M. J. Balley. The vectorization of ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, August 1985.

[26] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing. *The Visual Computer*, 5:109–119, 1989.

[27] R. Pulleyblank and J. Kapenga. The feasibility of a vlsi chip for ray tracing bicubic patches. *IEEE Computer Graphics and Applications*, 7:33–44, March 1987.

[28] M. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York, 1987.

[29] P. Sadayappan and F. Erçal. Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, C-36(12):1408–1423, December 1987.

[30] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics, part i: Fundamentals. *IEEE Computer Graphics and Applications*, 8(3):48–68, May 1988.

[31] H. Samet and R. E. Webber. Hierarchical data structures and algorithms for computer graphics, part ii: Applications. *IEEE Computer Graphics and Applications*, 8(4):59–75, July 1988.

[32] C. Upson and M. Keeler. The v-buffer: Visible volume rendering. *Computer Graphics*, 22(4):59–64, July 1988.

[33] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering.* Jones and Bartlett Publishers, London, 1992.

[34] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23:343–349, June 1980.

Vita

Veysi İşler received the B.S. degree in computer engineering from the Middle East Technical University, Ankara, Turkey, and the M.S. degree in computer engineering and information science from the Bilkent University, Ankara, Turkey, in 1987 and 1989, respectively. During his Ph.D. study at Department of Computer Engineering and Information Science at Bilkent University, he worked on spatial subdivision for parallel ray casting/tracing. His research interests include animation, rendering, visualization, and parallel processing for computer graphics.