

PARALLEL RENDERING ALGORITHMS FOR
DISTRIBUTED-MEMORY MULTICOMPUTERS

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
Tatjana Martynova Kurp
June 1997

PARALLEL RENDERING ALGORITHMS FOR DISTRIBUTED-MEMORY MULTICOMPUTERS

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

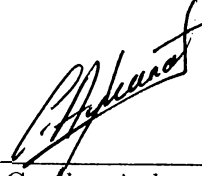
by
Tahsin Mertefe Kurç
June 1997

Tahsin Mertefe Kurç

QA
76.53
.K89
1997

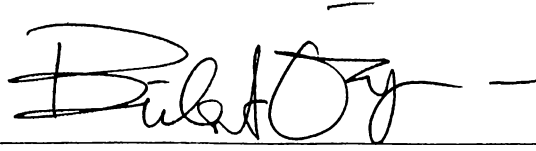
B037981

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.



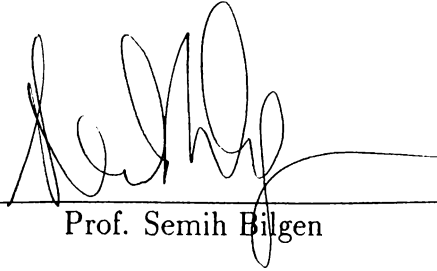
Assoc. Prof. Cevdet Aykanat (Supervisor)

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.



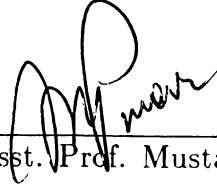
Prof. Bülent Özgüç (Co-supervisor)

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.

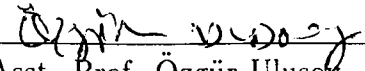


Prof. Semih Bilgen

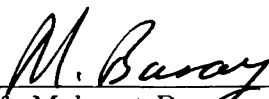
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of doctor of philosophy.


Asst. Prof. Mustafa Ç. Pinar

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.


Asst. Prof. Özgür Ulusoy

Approved by the Institute of Engineering and Science:


Prof. Mehmet Baray,
Director of the Institute of Engineering and Science

ABSTRACT

PARALLEL RENDERING ALGORITHMS FOR DISTRIBUTED-MEMORY MULTICOMPUTERS

Tahsin Mertefe Kurç

Ph.D. in Computer Engineering and Information Science

Supervisors:

Assoc. Prof. Cevdet Aykanat and Prof. Bülent Özgüç

June 1997

In this thesis, utilization of distributed memory multicomputers in *gathering radiosity*, *polygon rendering* and *volume rendering* is investigated.

In parallel gathering radiosity, the target issues are the parallelization of the computation of the form-factor matrix and solution phases on hypercube-connected multicomputers. Interprocessor communication in matrix computation phase is decreased by sharing the memory space between matrix elements and the scene data. A demand-driven algorithm is proposed for better computational load balance during calculation of form-factors. Gauss-Jacobi (**GJ**) iterative algorithm is used by all of the previous works in the solution phase. We apply more efficient Scaled Conjugate-Gradient (**SCG**) algorithm in the solution phase. Parallel algorithms were developed for **GJ** and **SCG** algorithms for hypercube-connected multicomputers. In addition, load balancing in the solution phase is investigated. An efficient data redistribution scheme is proposed. This

scheme achieves perfect load balance in matrix-vector product operations in the solution phase.

Object-space parallelism is investigated for parallel polygon rendering on hypercube-connected multicomputers. Briefly, in object-space parallelism, scene data is partitioned into disjoint sets among processors. Each processor performs the rendering of its local partition of primitives. After this *local rendering* phase, full screen partial images in each processor are merged to obtain the final image. This phase is called *pixel merging* phase. Pixel merging phase requires interprocessor communication to merge partial images. In this work, hypercube interconnection topology and message passing structure are exploited to merge partial images efficiently. Volume of communication in pixel merging phase is decreased by only exchanging local foremost pixels in each processor after local rendering phase. For this purpose, a modified scanline z-buffer algorithm is proposed for the local rendering phase. This algorithm avoids message fragmentation by storing local foremost pixels in consecutive memory locations. In addition, it eliminates initialization of z-buffer, which is a sequential overhead to parallel execution. For pixel merging phase, we propose two schemes referred to here as *pairwise exchange* scheme and *all-to-all personalized communication* scheme, which are suited to the hypercube topology. We investigate load balancing in pixel merging phase. Two heuristics, recursive subdivision and heuristic bin packing, were proposed to achieve better load balancing in pixel merging phase. These heuristics are adaptive such that they utilize the distribution of foremost pixels on the screen to subdivide the screen in the pixel merging phase.

Image-space parallelism is investigated for parallel volume rendering of unstructured grids. In image-space parallelism, the screen is subdivided into regions. Each processor is assigned one or more subregions. The primitives (e.g., tetrahedrons) in the volume data are distributed among processors according to screen subdivision and processor-subregion assignments. Then, each processor renders its local subregions. The target topic in this work is the adaptive subdivision of the screen. Adaptive subdivision issue has not been investigated in parallel volume rendering of unstructured grids before. Only some researchers utilized adaptive subdivision in parallel polygon rendering and ray tracing. In this work, several algorithms are proposed to subdivide the screen adaptively. The algorithms presented in this work can be grouped into two classes: 1-dimensional array

based algorithms and 2-dimensional mesh based algorithms. Among the 2-dimensional mesh based algorithms, graph partitioning based subdivision and Hilbert curve based subdivision algorithms are new approaches in parallel rendering field. An experimental comparison of the subdivision algorithms are performed on a common frame work. The subdivision algorithms were employed in the parallelization of a volume rendering algorithm, which is a polygon rendering based algorithm. In the previous works on parallel polygon rendering, only the number of primitives in a subregion were used to approximate the work load of the subregion. We experimentally show that this approximation is not enough. Better speedup values can be obtained by utilizing other criteria such as number of pixels, number of spans in a region. By utilizing these additional criteria, the speedup values are almost doubled.

ÖZET

ÇOK İŞLEMCİLİ DAĞITIK-HAFIZALI BİLGİSAYARLARDA PARALEL GÖRÜNTÜLEME ALGORİTMALARI

Tahsin Mertefe Kuş

Bilgisayar ve Enformatik Mühendisliği Doktora

Tez Yöneticileri:

Doç. Dr. Cevdet Aykanat and Prof. Bülent Özgüç

Haziran 1997

Bu tezde dağıtık hafızalı çok işlemcili bilgisayarların *ışma yönteminin toplama metodu*nda, *poligon görüntüleme*de ve *hacim görüntüleme*de kullanımı araştırılmıştır.

Toplama metodunda ele alınan temel konular durum-katsayı matrisinin hesaplanması ve çözüm adımının hiperküp bağlantılı çoklu bilgisayarlarda paralel olarak yapılmasıdır. Durum-katsayı matrisinin hesaplanmasında işlemciler arası veri aktarımı her işlemcideki hafızanın durum-katsayı matrisi ve ışma metoduyla görüntülenen ortamı oluşturan veriler arasında paylaşılması ile azaltılmıştır. İşlemcilerin daha verimli kullanılabilmesi için dinamik paylaşırma yöntemi uygulanmıştır. Çözüm aşamasında *Scaled Conjugate-Gradient* metodu başarılı bir şekilde uygulanmıştır. Gauss-Jacobi ve Scaled Conjugate-Gradient metodları için verimli paralel algoritmalar geliştirilmiştir. Durum-katsayı matrisinin hesaplanmasından sonra her işlemcide kalan sıfırdan farklı durum-katsayı değerlerinin işlemciler arasında tekrar dağıtılması ile hemen hemen ideal yük dağılımı sağlanmıştır.

Poligon görüntüleme konusunda yapılan çalışmalarda parça uzayında paralelleştirme yaklaşımı ele alınmıştır. Parça uzayında paralelleştirmede ortamı oluşturan parçalar işlemciler arasında dağıtılır. Her işlemci kendi parçalarının üzerinde görüntüleme algoritmalarını çalıştırır. Daha sonra her işlemcideki resimler birleştirilerek son resim ortaya çıkarılır. Bu çalışmada hiperküp bilgisayarında parça uzayında paralelleştirme algoritmaları geliştirilmiştir. Resimlerin birleştirilmesi sırasında işlemciler arasında iletişim hacmini azaltan verimli algoritmalar önerilmiştir. İşlemciler arasındaki mesajların kopuk kopuk olmasını önlemek için değiştirilmiş bir görüntüleme algoritması önerilmiştir.

Hacim görüntülemeye ise ekran uzayında paralelleştirme yaklaşımı araştırılmıştır. Bu yaklaşımda ekran uzayı işlemciler arasında bölünür. Her işlemci kendisine ait olan ekran parçası üzerinde görüntüleme algoritmasını çalıştırır. Ekranın bölünmesine göre hacim elemanları işlemciler arasında dağıtılır. Bu çalışmada, çeşitli ekran uzayında bölme yöntemleri incelendi ve geliştirildi. Bu yöntemler ekranı hacim elemanlarının ekrandaki dağılımlarına göre bölerek daha iyi yük dağılımı sağlar. Bu yöntemlerden çizge parçalamaya dayalı bölme ve Hilbert eğrisine dayalı bölme yeni yöntemlerdir. Bu yöntemler deneysel olarak karşılaştırılmıştır. Ayrıca, bu çalışmada incelenen ve geliştirilen yöntemler poligon görüntülemeye dayalı bir hacim görüntüleme algoritmasına başarı ile uygulanmıştır.

ACKNOWLEDGMENTS

I wish to express my deepest gratitude and thanks to Assoc. Prof. Dr. Cevdet Aykanat and Prof. Dr. Bülent Özgüç for their supervision, encouragement, and invaluable advice throughout the development of this thesis.

I am very grateful to Prof. Dr. Semih Bilgen, Asst. Prof. Dr. Özgür Ulusoy, and Asst. Prof. Dr. Mustafa Pınar for carefully reading my thesis, for their remarks and suggestions.

I would like to extend my sincere thanks to all of my friends for their morale support and encouragement during the thesis work. I would like to thank Egemen Tanin for his sequential volume rendering code. I owe special thanks to all members of our department for providing a pleasant environment for study.

Finally, my sincere thanks goes to my family for their endless morale support and patience.

This work was partially supported by the *Scientific and Technical Research Council of Turkey* (TÜBİTAK) under grants EEEAG-5 and EEEAG-160, *Intel Supercomputer Systems Division* under grant SSD100791-2, and the *Commission of the European Communities, Directorate General for Industry* under contract ITDC 204-82166.

Contents

1	Introduction	1
1.1	Gathering Radiosity	3
1.2	Polygon Rendering	5
1.3	Volume Rendering	5
1.4	Contributions of the Thesis	7
1.4.1	Parallel Gathering Radiosity	8
1.4.2	Parallel Polygon Rendering	9
1.4.3	Parallel Volume Rendering	10
1.5	Organization of the Thesis	11
2	Gathering Radiosity on Hypercubes	13
2.1	Gathering Radiosity	15
2.1.1	Form-Factor Computation Phase	16
2.1.2	Solution Phase	17
2.2	Previous Work on Parallel Gathering Radiosity	23
2.3	Intel's iPSC/2 Hypercube Multicomputer	24
2.4	Parallel Computation of the Form-Factor Matrix	27
2.4.1	Static Assignment	27
2.4.2	Demand-Driven Assignment Scheme	31
2.5	Parallel Solution Phase	32
2.5.1	Parallel Gauss-Jacobi Method	32
2.5.2	Parallel Scaled Conjugate-Gradient Method	35
2.5.3	A Parallel Renumbering Scheme	37

2.6	Load Balancing in the Solution Phase: Data Redistribution	39
2.6.1	A Parallel Data Redistribution Scheme	40
2.6.2	Avoiding the Extra Setup Time Overhead	42
2.7	Experimental Results	42
2.8	Conclusions	47
3	Polygon Rendering: Overview and Related Work	50
3.1	Sequential Polygon Rendering	50
3.1.1	Reading Environment Description	50
3.1.2	Lighting Calculations	52
3.1.3	Geometry Processing	52
3.1.4	Shading and Hidden-surface Removal	53
3.2	Previous Works on Parallel Polygon Rendering	57
3.2.1	A Taxonomy of Parallelism in Polygon Rendering on Distributed-Memory Multicomputers	57
3.2.2	Previous Works on Parallel Polygon Rendering	62
3.3	Discussion of Previous Works	71
4	Active Pixel Merging on Hypercubes	74
4.1	Some Definitions	75
4.2	The Parallel Algorithm	75
4.3	A Modified Scanline Z-buffer Algorithm	76
4.4	Pixel Merging on Hypercube Multicomputer	78
4.4.1	Ring Exchange Scheme	78
4.4.2	2-dimensional Mesh Exchange Scheme	80
4.4.3	K-dimensional Mesh Exchange Scheme	82
4.4.4	Pairwise Exchange Scheme	84
4.4.5	All-to-All Personalized Communication Scheme	85
4.4.6	Comparison of Pixel Merging Schemes	86
4.5	Load Balancing in Pixel Merging Phase	87
4.5.1	Recursive Adaptive Subdivision	87
4.5.2	Heuristic Bin Packing	88

4.6	Experimental Results on an iPSC/2 Hypercube Multicomputer	90
4.7	Results on a Parsytec CC System	94
4.8	Conclusions	98
5	Volume Rendering: Overview and Related Work	106
5.1	Nomenclature	107
5.2	Ray-casting Based Direct Volume Rendering	110
5.2.1	Point Location and View Sort Problems	111
5.2.2	Approaches to Solve Point Location and View Sort Problems . . .	114
5.3	Previous Works on Parallel Direct Volume Rendering of Unstructured Grids	119
5.4	Discussion of Previous Works on Parallel Volume Rendering of Unstructured Grids	123
6	Spatial Subdivision for Volume Rendering	125
6.1	Spatial Subdivision Algorithms	127
6.1.1	Horizontal Subdivision (HS)	128
6.1.2	Rectangular Subdivision (RS)	131
6.1.3	Recursive Rectangular Subdivision (RRS)	132
6.1.4	Mesh-based Adaptive Hierarchical Decomposition Scheme (MAHD)	134
6.1.5	Hilbert Curve Based Subdivision (HCS)	135
6.1.6	Graph Partitioning Based Subdivision (GS)	137
6.1.7	Redistributing the Primitives	140
6.2	Experimental Comparison of Subdivision Algorithms	141
6.3	Volume Rendering of Unstructured Grids: A Scanline Z-buffer Based Algorithm	149
6.4	The Parallel Algorithm	152
6.5	Experimental Results	154
6.6	Conclusions	156
7	Summary and Conclusions	161
7.1	Parallel Gathering Radiosity	161
7.2	Parallel Polygon Rendering	162

7.3	Parallel Volume Rendering	164
-----	-------------------------------------	-----

List of Figures

1.1	An example of computer graphics rendering.	2
2.1	Basic steps of the GJ method.	19
2.2	Basic steps of SCG method.	22
2.3	A 16 node hypercube multicomputer	25
2.4	(a) A ring embedding (b) A 2-dimensional mesh embedding into a 4-dimensional hypercube.	26
2.5	The node algorithm in pseudo-code for patch circulation scheme.	29
2.6	The node algorithm in pseudo-code for form-factor computation by storage sharing scheme.	30
2.7	Parallel GJ algorithm.	33
2.8	Global concatenate operation for a 3-dimensional hypercube.	34
2.9	Parallel SCG method.	36
2.10	Form-factor computation phase. (a) Execution times for different schemes on 16 processors. (b) Efficiency curves for different schemes.	44
2.11	The effect of the <i>assignment granularity</i> on the performance (execution time in seconds) of the demand driven scheme for $N = 886$, $P = 16$	45
2.12	Efficiency curves for the SCG method.	47
3.1	The polygon rendering pipeline.	51
3.2	A polygon with 5 vertices.	52
3.3	World and viewing coordinate systems.	53
3.4	The z-buffer array.	55

3.5	An example of image-space parallelism. The screen is partitioned and subregions are assigned to processors (P0 , P1 , P2 , P3).	58
3.6	An example of object-space parallelism.	59
4.1	Volume of communication on different meshes embedded on the hypercube of 16 processors for different scenes.	84
4.2	Extended span algorithm.	90
4.3	Comparison of RS with HBP. (a) Different number of processors for 2 POT scene, $A = 400 \times 400$. (b) Different screen resolutions and different scenes on 16 processors.	94
4.4	Volume of communication for (a) 2 POT scene on different processors, $A = 400 \times 400$. (b) $A = 400 \times 400$ and $A = 640 \times 640$ for different scenes on 16 processors.	95
4.5	Speedup figures for $A = 400 \times 400$. (a) 1 POT scene (b) 2 POT scene.	95
4.6	Speedup figures for $A = 640 \times 640$. (a) 1 POT scene (b) 2 POT scene.	96
4.7	The Parsytec CC system.	97
4.8	Rendering rates of algorithms on Parsytec CC system. (a) AAPC-HBP (b) ZBUF-EXC.	100
4.9	Speedup values achieved by the algorithms on Parsytec CC system. (a) AAPC-HBP (b) ZBUF-EXC.	100
4.10	Total volume of communication and concurrent volume of communication.	101
4.11	Rendered images of the scenes used in the experiments on iPSC/2. (a) 1 POT scene (b) 2 POT scene.	102
4.12	Rendered images of the scenes used in the experiments on iPSC/2. (a) 4 POT_1 scene (b) 4 POT_2 scene.	102
4.13	Rendered images of the scenes used in the experiments on iPSC/2. (a) 8 POT_1 scene (b) 8 POT_2 scene.	103
4.14	Rendered images of the scenes used in the experiments on the Parsytec CC system. (a) Teapot scene (102080 triangles, rendering time is 0.332 seconds on 16 processors) (b) Balls scene (157440 triangles, rendering time is 0.495 seconds on 16 processors).	104

4.15	Rendered images of the scenes used in the experiments on the Parsytec CC system. (a) Lattice scene (235200 triangles, rendering time is 0.7 seconds on 16 processors) (b) Rings scene (343200 triangles, rendering time is 0.821 seconds on 16 processors).	104
4.16	Rendered images of the scenes used in the experiments on the Parsytec CC system. (a) Tree scene (425776 triangles, rendering time is 0.576 seconds on 16 processors) (b) Mountain scene (524288 triangles, rendering time is 1.052 seconds on 16 processors).	105
5.1	A volumetric data set. Figure illustrates a 2-dimensional projection of the volume. (a) Volume is sampled at 3-dimensional space. Each small filled circle represents the sample points with 3-dimensional spatial coordinates. Dashed lines represent the boundaries of the volume. (b) Sample points are connected to form volume elements. A tetrahedral cell, which is formed by connecting four distinct sample points, is also illustrated. . .	108
5.2	Types of grids encountered in volume rendering.	110
5.3	Ray-casting based direct volume rendering.	111
5.4	Re-sampling phase of the ray-casting DVR. The color and opacity values at the sample point on the ray are calculated by finding the contributions of original sample points which form the cell. After re-sampling, sample points on the ray are composited to generate the color on the screen. . .	113
6.1	An example of horizontal subdivision for eight processors.	129
6.2	An example of rectangular division for 8 processors organized into 4 clusters and 2 processors in each cluster.	131
6.3	An example of recursive subdivision for eight processors.	132
6.4	An example of mesh-based adaptive hierarchical decomposition for eight processors. Mesh resolution is 8×8	136
6.5	Traversing of the 2-dimensional mesh with Hilbert curve and mapping of the mesh cells locations into one-dimensional array indices.	137
6.6	An example of Hilbert curve based subdivision for eight processors. Mesh resolution is 8×8	138

6.7	An example of graph partitioning based subdivision for eight processors. Mesh resolution is 8×8	139
6.8	The algorithm to classify the primitives at redistribution step of HS, RS, RRS, and MAHD algorithms.	141
6.9	The algorithm to classify primitives in HCS and GS algorithms.	142
6.10	Rendered images of the data sets used in the experiments. (a) Blunt fin (381548 triangles, rendering time is 6.27 seconds on 16 processors) (b) Post data (1040588 triangles, rendering time is 8.55 seconds on 16 processors).	143
6.11	Load balancing performance, based on the approximate load calculations, of the MAHD, HCS, and GS algorithms (a) Different mesh resolutions on 16 processors. (b) Different number of processors.	144
6.12	Percent increase in the number of primitives after primitive redistribution. Each value in the graph represents the percent increase in the total number of primitives for the mesh resolution the algorithm finds the best load balance, based on the estimated load distribution.	145
6.13	Load balancing performance, based on the actual primitive distribution, of the MAHD, HCS, and GS algorithms (a) Different mesh resolutions on 16 processors. (b) Different number of processors.	146
6.14	Percent increase in the number of primitives after primitive redistribution. Each value in the graph represents the percent increase in the total number of primitives for the mesh resolution the algorithm finds the best load balance, based on the actual distribution of primitives.	147
6.15	Execution time of MAHD, HCS, and GS for different mesh resolutions.	148
6.16	Load balance performance of all algorithms (HS, RS, RRS, MAHD, HCS, and GS) on different number of processors	149
6.17	Percent increase in the number of primitives after redistribution for all algorithms on different number of processors. Each value in the graph represents the percent increase in the total number of primitives for the mesh resolution the algorithm finds the best load balance, based on the actual distribution of primitives.	150

6.18	Execution time of all algorithms on different number of processors. For MAHD, HCS, and GS algorithms, the values represents the execution time of the respective algorithm for the mesh resolution the algorithm achieves the best load distribution.	151
6.19	Speedup for only rendering phase when only the number of triangles in a region is used to approximate work load in a region.	158
6.20	Speedup for rendering phase (step 4 of the parallel algorithm) when spans and pixels are incorporated into the subdivision algorithms.	158
6.21	Speedup values including the execution time of subdivision algorithms.	159
6.22	Errors due to bounding box approximation in calculating the number of spans when vertical divisions are allowed.	159
6.23	Rendering times in seconds. (a) Only rendering time excluding subdivision overhead. (b) Rendering time including subdivision overhead.	160

List of Tables

2.1	Relative performance results in parallel execution times (in seconds) of different parallel algorithms for the form-factor computation phase. N is the number of patches in the scene and M is the number of non-zero entries in the form-factor matrix.	43
2.2	Parallel execution times (in seconds) of various schemes in the solution phase (using GJ) along with the associated overheads. TOT is the total execution time including overheads, i.e. $TOT = (solution + preprocessing)$ time. N is the number of patches in the scene and M is the number of non-zero entries in the form-factor matrix.	46
2.3	Performance comparison of parallel <i>Gauss-Jacobi</i> and <i>Scaled Conjugate-Gradient</i> methods (1* denotes the estimated sequential timings). Timings are in seconds. N is the number of patches in the scene and M is the number of non-zero entries in the form-factor matrix.	49
4.1	Scene characteristics in terms of total number of pixels generated (TPG), number of triangles, and total number of winning pixels in the final picture (TPF) for different screen sizes.	91
4.2	Relative execution times (in milliseconds) of full z-buffer merging and PAIR-RS for N=400.	92
4.3	Comparison of execution times (in milliseconds) of several pixel merging schemes.	93
4.4	Number of triangles in the test scenes.	96

6.1 Dissection of execution time of each algorithm on different number of processors.	148
--	-----

Chapter 1

Introduction

Rendering in computer graphics can be described as the process of generating a 2-dimensional representation of a data set defined in 3-dimensional space. Input to this process is a set of *primitives* defined in a 3-dimensional coordinate system, usually called *world coordinate system*, and a *viewing* position and orientation also defined in the same world coordinate system. The primitives are objects, polygons, surfaces, or points connected in a predetermined way (as in volumetric data sets), which constitute the input data set. The viewing position and orientation define the orientation and location of the image-plane, which represents the computer screen. The output of the rendering process is a 2-dimensional picture of the data set on the computer screen. Figure 1.1 illustrates an example of computer graphics rendering with its input and output.

In this thesis, we investigate the utilization of distributed-memory multicomputers in three different fields of computer graphics rendering:

- *Realistic simulation of light propagation*: One of the challenging fields in computer graphics rendering is to model the light-object interactions and propagation of light in an environment realistically. Ray tracing [102] and radiosity [33] are two popular methods used in such applications. The target method in this thesis is the *gathering radiosity* [33] method.
- *Polygon rendering*: Algorithms and methods in polygon rendering field deal with producing realistic images of computer generated environments composed of polygons.

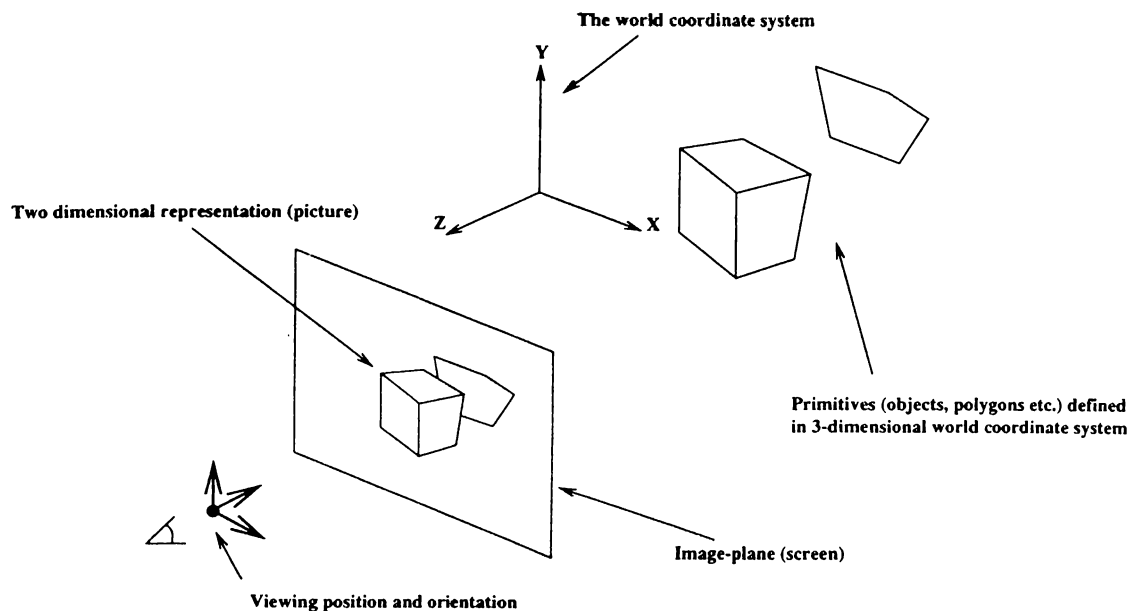


Figure 1.1: An example of computer graphics rendering.

- *Volume rendering*: Volume rendering techniques deal with visualization of scientific data sets composed of large amounts of numerical data values associated with points in 3-dimensional space. This thesis investigates methods for parallel rendering of *unstructured grids*, in which points are irregularly distributed in 3-dimensional space.

Realistic illumination models and shading methods, like gathering radiosity, require large memory space and computing power. Moreover, increased complexity of computer generated environments has added more memory space and more computing power requirements in polygon rendering. Similarly, techniques applied in volume rendering and huge size of data sets obtained in scientific applications require large memory space and high computing power. It is unlikely to meet increasing requirements of these fields on single processor machines with today's technology. whereas distributed-memory multi-computers can provide a cost-effective solution. Large memory space and high computing power requirements are met by connecting many processors with individual memories and using these processors simultaneously. Each processor in the architecture can perform computations asynchronously on different data values, thus providing a *flexible*

environment. Flexibility provides a cost-effective working environment for many applications of different nature and characteristics. Distributed-memory multicomputers can be upgraded and extended by adding more processors with individual memories to the environment, thus providing a *scalable* environment. Scalability provides an inherent power to meet increasing requirements of the applications. As the name implies, there is no shared global memory in distributed-memory architectures. Each processor has its own local memory, which cannot be directly accessed by other processors. Synchronization and data exchange between processors are carried out via exchanging messages over an interconnection network. Among many interconnection topologies, *rings*, *meshes*, *hypercubes*, and *multistage switch based networks* are the most commonly used network topologies.

In this chapter, brief overviews of *gathering radiosity*, *polygon rendering*, and *volume rendering* are given. Following the overviews, contributions of the thesis work are presented. Organization of the thesis is given in the last section.

1.1 Gathering Radiosity

Given a description of the environment, producing a realistic image of the environment on the computer is accomplished in three basic steps: (1) – *reading the description of the environment and converting the description into appropriate form to apply rendering algorithms*, (2) – *simulating the propagation of light in the environment*, (3) – *displaying the environment on the computer screen*.

At the first step, the description of the environment to be rendered is read into the computer. The description of the environment is converted into appropriate form that is suitable for algorithms to simulate light propagation and to display the environment. For example, surfaces and objects descriptions should be converted into polygons to apply polygon rendering algorithms.

Next step is to simulate the propagation of light in the environment and light-object interactions. Various methods are used in computer graphics rendering [95, 96, 77]. Simple methods, such as Phong method [68], only simulate the interaction of light, coming directly from light sources, with the objects in the environment. These methods

result in moderate realism in the images because the contributions of light reflected from other objects in the environment are not considered. More realistic and complex methods account for the reflected and refracted light as well. There are two methods, called *ray tracing* [102] and *radiosity* [33], which are widely used for accurately simulating propagation of light in an environment.

The radiosity method accounts for the diffuse inter-reflections between the surfaces in a diffuse environment. There are two approaches to radiosity, *progressive refinement* [15] and *gathering* [33] methods. Gathering is a very suitable approach for investigating lighting effects within a closure. In this method, every surface and object constituting the environment is discretized into small patches (polygons), which are assumed to be perfect diffusers. The algorithm calculates the radiosity value of each patch in the scene. Initially, all patches, except for light sources, have zero initial radiosity values. The light sources are also treated as patches. The algorithm consists of three successive computational phases: *form-factor computation phase*, *solution phase* and *rendering phase*.

The form-factor matrix is computed and stored in the first phase. In an environment discretized into N patches, the radiosity b_i of each patch “i” is computed as follows:

$$b_i = e_i + r_i \sum_{j=1}^N b_j F_{ij} \quad (1.1)$$

where e_i and r_i denote the initial radiosity and reflectivity values, respectively, of patch “i” and the form-factor F_{ij} denotes the fraction of light that leaves the patch “i” and incident on patch “j”. The value of F_{ij} depends on the geometry of the scene and it is constant as long as the geometry of the scene remains unchanged. The F_{ii} values are taken to be zero for convex patches. This linear system of equations can be represented in matrix form as follows

$$\mathbf{Cb} = (\mathbf{I} - \mathbf{RF})\mathbf{b} = \mathbf{e} \quad (1.2)$$

where, \mathbf{R} is the diagonal reflectivity matrix, \mathbf{b} is the radiosity vector to be calculated, \mathbf{e} is the vector representing the self emission (initial emission) values of patches, and \mathbf{F} is the form-factor matrix.

In the second phase, a linear system of equations is solved for each color-band (e.g. red, green, blue) to find the radiosity values of all patches for these colors. In the last

phase, results are rendered and displayed on the screen using the radiosity values of the patches computed in the second phase. The radiosity values are transformed into color values for shading the polygons. Conventional polygon rendering methods [95, 77] (e.g. Gouraud shading, z-buffer algorithm) are used in the last step to display the results.

1.2 Polygon Rendering

As noted in the previous section, the last step of realistic image generation is to display the environment on the computer screen. A pipeline of operations is applied to transform polygons from 3-dimensional space to 2-dimensional screen space, perform smooth shading of the polygons, and perform hidden-surface removal to give realism to the image produced. Light-polygon interactions and shading of the polygons can also be done concurrently with hidden-surface removal if simple methods to calculate light-object interactions are used. Hidden-surface removal is a kind of sorting operation [86] to determine the visible parts of the polygons. Polygons are sorted by their distance to the screen. The overhead of sorting is decreased by utilizing some kind of coherency existing in the environment. Among many algorithms, *z-buffer* and *scanline z-buffer* algorithms are more popular due to wider range of applications and better utilization of coherency. These algorithms are called image-space algorithms since hidden-surface removal, hence sorting, is performed at pixel locations on the screen. In order to accomplish this, polygons are projected onto the screen and distance values are generated for screen coordinates covered by the projection of the polygon. Hidden-surface removal at a pixel location is done by comparing the distance values generated at the pixel location. These algorithms utilize image-space coherency to calculate the distance values at pixel locations. Calculation of distance value from one pixel to the next is done via incremental operations.

1.3 Volume Rendering

Visualization of scientific data aims at displaying vast amount of numerical data obtained from engineering simulations or gathered by scanning real physical entities by

advanced scan devices. Visualization of *volumetric data sets*, in which numerical values are obtained at *sample points* with 3-dimensional spatial coordinates in a volume, is referred to here as *volume rendering*. Sample points in these sets form a 3-dimensional grid superimposed on the volume. In this grid, sample points are connected to other sample points in a predetermined way to form volume elements, referred to here as *cells*. A sample point may be shared by many cells. In addition, a cell may share a face with other cells, forming a connectivity relation between volume elements. In volumetric data sets, two types of grids are commonly encountered. In *structured grids*, the sample points are regularly distributed in the volume. There exists implicit and regular connectivity between volume elements. This type of grids are most common in medical imaging applications. In *unstructured grids*, the sample points are distributed irregularly in the 3-dimensional space. There exists irregular connectivity between volume elements if a connectivity relation exists at all. Unstructured grids are commonly used in engineering simulations (e.g., computational fluid dynamics).

Volumetric data sets are rendered by finding the contribution of sample points to the pixels on the screen. These contributions are determined via processing the volume elements. Each of these contributions are transformed into color values to display the volume. Among many techniques in volume rendering, *ray-casting based direct volume rendering* [54, 92], which is the basis of research on parallel volume rendering in this thesis, has become very popular. Direct volume rendering (DVR) describes the process of visualizing the volume data without generating an intermediate geometrical representation such as *isosurfaces*. In ray-casting based direct volume rendering, rays are cast from pixel locations and traced in the volume. During the traversal in the volume, sample points are taken along the ray. The contribution of the volume element that contains the sample point is calculated. Then, these values at each sample point on the ray are composited in a predetermined order (front-to-back or back-to-front) to obtain the contribution at the pixel. Determining the volume element that contains the sample point is called *point location* problem and compositing the contributions in a predetermined order is called *view sort* problem. Resolving point location and view sort problems is a crucial issue that closely affects the performance of the rendering algorithm. Handling

point location and view sort problems in structured grids are easy due to regular distribution of sample points and implicit regular connectivity between volume elements. On the other hand, irregular distribution of sample points and irregular connectivity relation between volume elements (if it exists) make the point location and view sort problems much more difficult to handle in unstructured grids.

Application of Polygon Rendering Algorithms in Volume Rendering

Although polygon rendering and volume rendering form two diverse application areas of computer graphics in many aspects, techniques and algorithms used in one field can easily be adapted to resolve the problems in the other field.

As is stated, ray-casting based direct volume rendering algorithms should resolve the point location and view sort problems efficiently as these problems affect the performance directly. In many application, these problems reduce to finding the intersection of ray with respective volume elements. These intersections are then sorted in increasing distance from the screen so that composition of contributions of sample points is done correctly.

In polygon rendering, image-space hidden-surface removal algorithms such as z-buffer and scanline z-buffer actually perform a similar sorting of the object database to perform hidden-surface removal correctly. In addition, polygons are rasterized (or scan-converted) to generate color and distance values for the pixels covered by the polygon. This rasterization corresponds, in a sense, to finding the intersection of the polygon with the rays cast from those pixel locations. Image-space hidden-surface removal algorithms utilize image-space coherency to decrease the overheads of sorting and rasterization. Such a coherency also exists in ray-casting based DVR applications. Therefore, ray-casting based DVR can benefit from the application of polygon rendering algorithms since the basic problems are almost the same.

1.4 Contributions of the Thesis

In this thesis, utilization of distributed memory multicomputers in three fields, which are gathering radiosity, polygon rendering and volume rendering, in computer graphics

is investigated. This section presents the contributions of the thesis work.

1.4.1 Parallel Gathering Radiosity

Parallelization of form-factor matrix computation and solution phases of the gathering radiosity are the key issues in this work. The contributions of the thesis work in these issues are the following.

- In parallel computation of the form-factor matrix, several algorithms were developed. Interprocessor communication is decreased by sharing the memory space between matrix elements and the objects in the scene. A demand-driven algorithm is proposed to achieve better load balance among processors in form-factor computations. Our demand-driven approach is different from [12, 13]. Unlike their approach, we avoid re-distribution of matrix rows after matrix is calculated by not doing a *conceptual* partitioning of patches among processors. However, our scheme necessitates two-level indexing in matrix-vector product operations in the solution phase. A parallel re-numbering scheme is proposed to eliminate two-level indexing.
- All previous works used Gauss-Jacobi (**GJ**) iterative algorithm in the solution phase. We apply more efficient Scaled Conjugate-Gradient (**SCG**) algorithm in the solution phase. The non-symmetric coefficient matrix is converted to a symmetric matrix to apply **SCG**. This conversion is done without perturbing the sparsity structure of the matrix.
- Parallel algorithms were developed for **GJ** and **SCG** algorithms for hypercube-connected multicomputers. In order to achieve better load balance in the solution phase, an efficient data redistribution scheme is proposed. This scheme achieves perfect load balance in matrix-vector product operations in the solution phase. We obtain high efficiency values in the solution phase using **SCG** with data redistribution.

A paper version of this work will appear in [50].

1.4.2 Parallel Polygon Rendering

Object-space parallelism (Section 3.2.1) is investigated for parallel polygon rendering on hypercube-connected multicomputers. Briefly, in object-space parallelism, scene data is partitioned into disjoint sets among processors. Each processor performs the rendering of its local partition of primitives. This phase of rendering is referred to as *local rendering* phase. Then, full screen partial images in each processor are merged to obtain the final image. This phase is called *pixel merging* phase. The pixel merging phase necessitates interprocessor communication to merge partial images. In this work, hypercube interconnection topology and message passing structure is exploited in pixel merging phase. The contributions in this thesis are the following.

- Volume of communication in pixel merging phase is decreased by only exchanging local foremost pixels in each processor after local rendering phase.
- A modified scanline z-buffer algorithm is proposed for local rendering phase. The nice features of this algorithm are: It avoids message fragmentation by storing local foremost pixels in consecutive memory locations efficiently. In addition, it eliminates initialization of scanline z-buffer for each scanline on the screen. Initialization of z-buffer introduces a sequential overhead to parallel rendering.
- For pixel merging phase, we propose two schemes referred to here as *pairwise exchange* scheme and *all-to-all personalized communication* (AAPC) scheme, which are suited to the hypercube topology. Pairwise exchange scheme involves minimum number of communication steps, but it has memory-to-memory copy overheads. All-to-all personalized communication scheme eliminates these overhead by increasing the number of communication steps. Our AAPC scheme differs from 2-phase direct pixel forwarding of Lee [53]. Our algorithm is 1-phase algorithm, i.e., pixels are transmitted to destination processors in a single communication phase. Hence, our algorithm avoids the intermediate z-buffering in [53] totally.
- All of the processors are utilized actively throughout the pixel merging phase by exploiting the interconnection topology of hypercube and by dividing the screen among processors.

- We investigate load balancing in pixel merging phase. Two heuristics, recursive subdivision and heuristic bin packing, were proposed to achieve better load balancing in pixel merging phase. These heuristics are adaptive in that they utilize the distribution of foremost pixels on the screen to subdivide the screen for the pixel merging phase.

Most of the research work was performed on Intel's iPSC/2 hypercube multicomputer. Recently, the AAPC scheme with heuristic bin packing algorithm was ported to Parsytec's CC system with PowerPC processors. In the current implementation, a hypercube topology is assumed and the topology of CC system is not exploited. Our preliminary results on the CC system achieves rendering rates of 300K – 700K triangles/sec on 16 processors.

An earlier version of the parallel polygon rendering work appears in [51].

1.4.3 Parallel Volume Rendering

In volume rendering field, image-space parallelism (Section 3.2.1) for parallel volume rendering of unstructured grids is investigated. In image-space parallelism, the screen is subdivided into regions. Each processor is assigned one or more subregions. The primitives (e.g., tetrahedrals) in the volume data are distributed among processors according to screen subdivision and processor-subregion assignments. Then, each processor renders its local subregions. The contributions in this thesis are the following.

- Main topic in this work is the adaptive subdivision of screen for better load balance. Adaptive subdivision issue has not been investigated before in parallel volume rendering of unstructured grids. Only some researchers utilized adaptive subdivision in parallel polygon rendering [76, 99, 65, 26] and in ray tracing/casting [5]. The algorithms presented in this work can be grouped into two classes: 1-dimensional array based algorithms and 2-dimensional mesh based algorithms.
- Among the 2-dimensional mesh based algorithms, graph partitioning based subdivision and Hilbert curve based subdivision algorithms are new approaches in parallel rendering.

- An experimental comparison of the subdivision algorithms is performed on a common frame work.
- The subdivision heuristics are employed in parallelization of a volume rendering algorithm. The sequential volume rendering algorithm is based on Challinger's work [9, 10]. This algorithm is basically a polygon rendering based algorithm. It requires volume elements composed of polygons and utilizes a scanline z-buffer approach to resolve point location and view sort problems. In the previous works on parallel polygon rendering, only the number of primitives in a subregion were used to approximate the work load of the subregion. We experimentally show that this approximation is not enough. Better speedup values can be obtained by utilizing other criteria such as number of pixels and number of spans in a region. By utilizing these additional criteria, the speedup values are almost doubled.

An earlier version of the parallel volume rendering work is published in [89].

1.5 Organization of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2, parallel implementation of form-factor computation and solution phases of gathering radiosity on hypercube-connected multicomputers is presented. A brief description of iPSC/2 hypercube multicomputer, an overview of gathering radiosity and previous work on parallel gathering radiosity are also included in this chapter.

Chapter 3 presents an overview of sequential polygon rendering. In addition, a taxonomy of parallelism in polygon rendering is introduced. Previous works, classified with respect to this taxonomy, are summarized in this chapter.

In Chapter 4, an object-space parallel algorithm for polygon rendering on hypercube multicomputers is presented. Several schemes for efficient implementation of local rendering and pixel merging phases are described.

An overview of volume rendering for unstructured grids is presented in Chapter 5. Previous works on parallel volume rendering of unstructured grids are summarized in this chapter.

Spatial subdivision algorithms, developed in this thesis, for image-space parallel volume rendering are described in Chapter 6.

Chapter 2

Gathering Radiosity on Hypercubes

Realistic synthetic image generation by computers has been a challenge for many years in the computer graphics field. Realistic synthetic image generation requires the accurate calculation and simulation of light propagation and global illumination effects in an environment. The *radiosity* method [33] is one of the techniques to simulate the light propagation in a closed environment. Radiosity accounts for the diffuse inter-reflections between the surfaces in a diffuse environment. There are two approaches to radiosity, *progressive refinement* [15] and *gathering* [33] methods. The gathering method (the term *radiosity method* will also be used interchangeably to refer to gathering method) consists of three successive computational phases: *form-factor computation phase*, *solution phase* and *rendering phase*. The form-factor matrix is computed and stored in the first phase. In the second phase, a linear system of equations is formed and solved for each color-band (e.g. red, green, blue) to find the radiosity values of all patches for these colors. In the last phase, results are rendered and displayed on the screen using the radiosity values of the patches computed in the second phase. Conventional rendering methods [95, 77] (e.g. Gouraud shading, z-buffer algorithm) are used in the last phase to display the results.

Gathering is a very suitable approach for investigating lighting effects within a closed environment. For such applications, the locations of the objects and light sources in the scene usually remain fixed while the intensity and color of light sources and/or reflectivity of surfaces change in time. The linear system of equations are solved many times to

investigate the effects of these changes. Therefore, efficient implementation of the solution phase is important for such applications. Although gathering is excellent for some applications in realistic image generation, it requires high computing power and large memory storage to hold the scene data and computation results. As a result, applications of the method on conventional uniprocessor computers for complex environments can be far from being practical due to high computation and memory costs.

In this chapter, parallelization of the first two phases of the gathering method is investigated for hypercube-connected multicomputers. In parallel computation of form-factor matrix, several algorithms were developed. Interprocessor communication is decreased by sharing the memory space between matrix elements and the objects in the scene. A demand-driven algorithm is proposed to achieve better load balance among processors in form-factor computations. Our demand-driven approach is different from [12, 13]. We do not perform a *conceptual* partitioning of patches among processors. Thus, matrix rows are not redistributed after the matrix is calculated. However, our scheme necessitates two-level indexing in matrix-vector product operations in the solution phase. An efficient parallel re-numbering scheme is proposed to eliminate the two-level indexing.

The previous works [12, 13, 67, 73] utilized Gauss-Jacobi (**GJ**) iterative algorithm in the solution phase. We apply the more efficient Scaled Conjugate-Gradient (**SCG**) algorithm in the solution phase. The non-symmetric coefficient matrix is converted into a symmetric matrix to apply **SCG**. This conversion is done without perturbing the sparsity structure of the matrix. Parallel algorithms were developed for **GJ** and **SCG** algorithms for hypercube-connected multicomputers. In addition, load balancing in the solution phase is investigated. An efficient data redistribution scheme is proposed. This scheme achieves perfect load balance in matrix-vector product operations in the solution phase. We obtain high efficiency values in the solution phase using **SCG** with data redistribution.

The organization of this chapter is as follows. Section 2.1 describes the computational requirements and the methods used in the form-factor computation and solution phases. The proposed **SCG** algorithm is described in this section as well. Section 2.2 briefly summarizes the existing work on the parallelization of the gathering radiosity method. Section 2.4 presents the parallel algorithms developed for the form-factor computation

phase. The parallel algorithms developed for the solution phase are presented and discussed in Section 2.5. Load balancing in the solution phase and a data redistribution scheme are discussed in Section 2.6. Finally, experimental results on a 16-node Intel iPSC/2 hypercube multicomputer are presented and discussed in Section 2.7.

2.1 Gathering Radiosity

Radiosity is based on the energy equilibrium within a closure. In this method, every surface and object constituting the environment is discretized into small patches. Each patch is assumed to be a perfect diffuser or an *ideal Lambertian* surface. The algorithm calculates the radiosity value of each patch in the scene. The radiosity value of a patch is defined to be the amount of light leaving that patch in equilibrium state. It is a function of emitted and reflected light from the patch. Initially, all patches have zero initial radiosity values. The light sources are also treated as patches except they possess non-zero initial radiosity values. In an environment discretized into N patches, the radiosity b_i of each patch “i” is computed as follows:

$$b_i = e_i + r_i \sum_{j=1}^N b_j F_{ij} \quad (2.1)$$

where e_i and r_i denote the initial radiosity and reflectivity values, respectively, of patch “i” and the form-factor F_{ij} denotes the fraction of light that leaves the patch “i” and incident on patch “j”. The value of F_{ij} depends on the geometry of the scene and it is constant as long as the geometry of the scene remains unchanged. This linear system of equations can be represented in matrix form as follows:

$$\begin{bmatrix} 1 - r_1 F_{11} & -r_1 F_{12} & & -r_1 F_{1N} \\ -r_2 F_{21} & 1 - r_2 F_{22} & \dots & -r_2 F_{2N} \\ & & \ddots & \\ -r_N F_{N1} & -r_N F_{N2} & \dots & 1 - r_N F_{NN} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \quad (2.2)$$

The F_{ii} values are taken to be zero for convex patches. Assuming $F_{ii} = 0$, the coefficient matrix in Eq. (2.2) can further be decomposed into three matrices as

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} - \begin{bmatrix} r_1 & 0 & 0 & \dots & 0 \\ 0 & r_2 & 0 & \dots & 0 \\ 0 & 0 & r_3 & \dots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \dots & r_N \end{bmatrix} \begin{bmatrix} 0 & F_{12} & F_{13} & \dots & F_{1N} \\ F_{21} & 0 & F_{23} & \dots & F_{2N} \\ F_{31} & F_{32} & 0 & \dots & F_{3N} \\ \vdots & \vdots & \vdots & & \vdots \\ F_{N1} & F_{N2} & F_{N3} & \dots & 0 \end{bmatrix}$$

I
R
F

Hence, Eq. (2.2) can be re-written as follows:

$$\mathbf{C}\mathbf{b} = (\mathbf{I} - \mathbf{R}\mathbf{F})\mathbf{b} = \mathbf{e} \quad (2.3)$$

where, \mathbf{R} is the diagonal reflectivity matrix, \mathbf{b} is the radiosity vector to be calculated, \mathbf{e} is the vector representing the self emission (initial emission) values of patches, and \mathbf{F} is the form-factor matrix.

2.1.1 Form-Factor Computation Phase

An approximate method to calculate the form-factors is proposed in [16], called the *hemi-cube* method. In this method, a discrete hemi-cube is placed around the center of a patch. Each face of the hemi-cube is divided into small squares (surface squares). A typical hemi-cube is composed of 100x100x50 such squares. Each square “s” corresponds to a delta form-factor ($\Delta f(s)$), which is a function of the area of the square, and the displacement of the square in x,y or y,z directions depending on the square “s” being located on the top face or side faces of the hemi-cube, respectively.

After allocating a hemi-cube over a patch “i”, all other patches in the environment are projected onto the hemi-cube for hidden patch removal. The patches are passed through a projection pipeline consisting of visibility test, clipping, perspective projection, and scan conversion. This projection pipeline is analogous to a z-buffer algorithm except for the fact that patch numbers are recorded for each allocated hemi-cube surface square in addition to z values. Then, each square “s” allocated by patch “j” contributes $\Delta f(s)$ to the form-factor F_{ij} between patches “i” and “j”. At the end of this process i^{th} row of the form-factor matrix \mathbf{F} is constructed. This process is repeated for all patches in the

environment in order to construct the whole \mathbf{F} matrix. Sum of the form-factor values in each row of the \mathbf{F} matrix is equal to 1 by definition.

The \mathbf{F} matrix is a sparse matrix because a patch does not see all the patches in the environment due to the occlusions. Almost 60%-85% of the \mathbf{F} matrix elements are observed to be zero in the test scenes. In order to reduce the memory requirements, \mathbf{F} matrix is stored in compressed form. Space is allocated for only non-zero elements of the matrix dynamically during the form-factor computation phase. Each element of a row of the matrix is in the form $[column-id, value]$. The *column-id* indicates the j index of an F_{ij} value in the i^{th} row.

2.1.2 Solution Phase

In this phase, the linear system of equations (Eq. (2.3)) is solved for each color-band. Methods for solving such linear system of equations can be grouped as direct methods and iterative methods. Direct methods such as *Gaussian elimination* and *LU factorization* [32] disturb the original sparsity of the coefficient matrices during the factorization. Furthermore, direct methods necessitate maintaining a coefficient matrix and two factor matrices for each color matrix for lighting simulations. As a result, direct methods require excessive memory for the solution phase of the radiosity method.

Iterative methods start from an initial vector \mathbf{b}^0 and iterate until a predetermined convergence criterion is reached. The sparsity of the coefficient matrix is preserved through out the iterations. Maintaining only the form-factor matrix \mathbf{F} suffices in the formulations of the iterative methods proposed in this work. Experimental results demonstrate that iterative methods converge quickly to acceptable accuracy values in the solution phase of the radiosity method. Furthermore, iterative methods are in general more suitable for parallelization than direct methods. Hence, direct methods are not considered in this work.

Three popular iterative methods widely used for solving linear system of equations are *Gauss-Jacobi (GJ)*, *Gauss-Seidel (GS)*, and *Conjugate-Gradient (CG)* [32]. The computational complexity of a **GS** iteration is exactly equal to that of **GJ** scheme. In general, **GS** scheme converges faster than the **GJ** scheme. Unfortunately, the **GS** scheme is inherently sequential and hence it is not suitable for parallelization. Thus, only **GJ**

and **CG** schemes are described and investigated for parallelization in this work.

Gauss-Jacobi Method

In the **GJ** method, the coefficient matrix **C** is decomposed as $\mathbf{C} = \mathbf{D} - \mathbf{L} - \mathbf{U}$ where **D**, **L** and **U** are the diagonal, lower triangular and upper triangular parts of **C** respectively. Then, the iteration equation can be represented in matrix notation as

$$\mathbf{b}^{k+1} = \mathbf{D}^{-1}((\mathbf{L} + \mathbf{U})\mathbf{b}^k + \mathbf{e}). \quad (2.4)$$

Since $\mathbf{C} = \mathbf{I} - \mathbf{RF}$ and $F_{ii} = 0$ for all $i = 1, 2, \dots, N$, we have $\mathbf{D} = \mathbf{I}$ and $\mathbf{L} + \mathbf{U} = \mathbf{RF}$. Hence, the iteration equation for the solution phase of the radiosity becomes

$$\mathbf{b}^{k+1} = \mathbf{RFb}^k + \mathbf{e}. \quad (2.5)$$

Recalling that the linear system of equations is to be constructed and solved for each color band, the **GJ** iteration equations for different colors can be re-written as

$$\mathbf{b}^{k+1}(r, g, b) = \mathbf{R}(r, g, b)\mathbf{Fb}^k(r, g, b) + \mathbf{e}(r, g, b). \quad (2.6)$$

Note that, it suffices to store only the diagonals of the diagonal **R** matrix. Hence, matrix and vector will be used interchangeably to refer to diagonal matrix. Therefore, Eq. (2.6) clearly illustrates that the **GJ** algorithm necessitates storing only the original **F** matrix and the reflectivity vector for each color in the solution phase. In order to minimize the computational overhead during the iterations due to this storage scheme, the matrix product **RF**, which takes $\Theta(M)$ time, should be avoided in the implementation, where M denotes the total number of non-zero entries in the **F** matrix. That is, the first term in the right-hand-side of the Eq. (2.5) should be computed as a sequence of two matrix-vector products $\mathbf{x} = \mathbf{Fb}$ and \mathbf{Rx} , which take $\Theta(M)$ and $\Theta(N)$ times, respectively. Since $M = O(N^2)$ is asymptotically larger than N , this computational overhead is negligible. The algorithm for **GJ** method is given in Fig. 2.1. The computational complexity of an individual **GJ** iteration is

$$T_{GJ} \approx (2M + 6N)t_{calc}. \quad (2.7)$$

Here, scalar addition, multiplication and absolute value operations are assumed to take the same amount of time t_{calc} .

Initially, choose \mathbf{b}^0
 for $k = 1, 2, 3, \dots$

1. form $\mathbf{b}^{k+1} = \mathbf{R}\mathbf{F}\mathbf{b}^k + \mathbf{e}$ as
 $\mathbf{x} = \mathbf{F}\mathbf{b}^k$; $\mathbf{y} = \mathbf{R}\mathbf{x}$; $\mathbf{b}^{k+1} = \mathbf{y} + \mathbf{e}$
 2. $\mathbf{r}^k = \mathbf{b}^{k+1} - \mathbf{b}^k$
 3. check $\text{Norm}(\mathbf{r}^k) / \max(\mathbf{b}^k) < \epsilon$
 where $\text{Norm}(\mathbf{r}^k) = \sum_{i=1}^N |r_i^k|$ and $\max(\mathbf{b}^k) = \max(|b_i^k|)$
-

Figure 2.1: Basic steps of the **GJ** method.

The convergence of the **GJ** method is guaranteed if the coefficient matrix is diagonally dominant. In radiosity, the coefficient matrix $\mathbf{C} = \mathbf{I} - \mathbf{R}\mathbf{F}$ satisfies diagonal dominance since $\sum_{j=1}^N F_{ij} = 1$, $F_{ii} = 0$ and $0 < r_i < 1$ (for each color band) for each row “i”.

Scaled Conjugate-Gradient Method

The **CG** method [39] is an optimization technique, iteratively searching the space of vectors \mathbf{b} to minimize the objective function $f(\mathbf{b}) = 1/2 \langle \mathbf{b}, \mathbf{C}\mathbf{b} \rangle - \langle \mathbf{e}, \mathbf{b} \rangle$ where $\mathbf{b} = [b_1, \dots, b_N]^t$, $f : R^N \rightarrow R$ and $\langle \cdot, \cdot \rangle$ denotes the inner product of two vectors. If the coefficient matrix \mathbf{C} is a symmetric and positive-definite matrix the objective function defined above is a strictly convex function and has a global minimum where its gradient vector vanishes, i.e. $\nabla f(\mathbf{b}) = \mathbf{C}\mathbf{b} - \mathbf{e} = 0$, which is also the solution to Eq. (2.3). The **CG** algorithm seeks this *global* minimum by finding in turn the local minima along a series of lines, the directions of which are given by vectors $\mathbf{p}_0, \mathbf{p}_1, \dots$ in an N-dimensional space.

As is mentioned earlier, the convergence of the **CG** method is guaranteed only if the coefficient matrix \mathbf{C} is symmetric and positive-definite. However, the original coefficient matrix is not symmetric since $c_{ij} = r_i F_{ij} \neq r_j F_{ji} = c_{ji}$. Therefore, the **CG** method cannot be used in the solution phase using the original \mathbf{C} matrix as is also mentioned in [67]. However, the reciprocity relation $A_i F_{ij} = A_j F_{ji}$ between the form factor values of the patches can be exploited to transform the original linear system of equations in

Eq. (2.3) into

$$\mathbf{S}\mathbf{b} = \mathbf{D}\mathbf{e} \quad (2.8)$$

with a symmetric coefficient matrix $\mathbf{S} = \mathbf{D}\mathbf{C}$ where \mathbf{D} is a diagonal matrix $\mathbf{D} = \text{diag}[A_1/r_1, A_2/r_2, \dots, A_N/r_N]$. Note that matrix \mathbf{S} is symmetric since $s_{ij} = A_i F_{ij} = A_j F_{ji} = s_{ji}$ for $j \neq i$. The i^{th} row of the matrix \mathbf{S} has the following structure

$$S_{i*} = [-A_i F_{i1}, \dots, -A_i F_{i,i-1}, A_i/r_i, -A_i F_{i,i+1}, \dots, -A_i F_{iN}]$$

for $i = 1, 2, \dots, N$. Therefore, matrix \mathbf{S} preserves diagonal dominance since $\sum_{i=1}^N F_{ij} = 1$ and $0 < r_i < 1$ (for each color band) for each row “i”. Thus, the coefficient matrix \mathbf{S} in the transformed system of equations (Eq. (2.8)) is positive-definite since diagonal dominance of a matrix ensures its positive-definiteness.

The convergence rate of the **CG** method can be improved by preconditioning. In this work, simple yet effective *diagonal scaling* is used for preconditioning the coefficient matrix \mathbf{S} . In this preconditioning scheme, rows and columns of the coefficient matrix \mathbf{S} are individually *scaled* by its diagonal $\mathbf{D} = \text{diag}[A_1/r_1, \dots, A_N/r_N]$. Therefore, the **CG** algorithm is applied to solve the following linear system of equations

$$\tilde{\mathbf{S}}\tilde{\mathbf{b}} = \tilde{\mathbf{e}} \quad (2.9)$$

where $\tilde{\mathbf{S}} = \mathbf{D}^{-1/2}\mathbf{S}\mathbf{D}^{-1/2} = \mathbf{D}^{-1/2}\mathbf{D}\mathbf{C}\mathbf{D}^{-1/2} = \mathbf{D}^{1/2}\mathbf{C}\mathbf{D}^{-1/2}$ has unit diagonals, $\tilde{\mathbf{b}} = \mathbf{D}^{1/2}\mathbf{b}$ and $\tilde{\mathbf{e}} = \mathbf{D}^{-1/2}\mathbf{D}\mathbf{e} = \mathbf{D}^{1/2}\mathbf{e}$. Thus, the right-hand side vector $\mathbf{D}\mathbf{e}$ in Eq. (2.8) is also scaled and $\tilde{\mathbf{b}}$ must be scaled back at the end to obtain the original solution vector \mathbf{b} (i.e. $\mathbf{b} = \mathbf{D}^{-1/2}\tilde{\mathbf{b}}$). The eigenvalues of the scaled coefficient matrix $\tilde{\mathbf{S}}$ (in Eq. (2.9)) are more likely to be grouped together than those of the unscaled matrix \mathbf{S} (in Eq. (2.8)), thus resulting in a better condition number.

The entries of the scaled coefficient matrix $\tilde{\mathbf{S}}$ are of the following structure:

$$\tilde{s}_{ij} = \begin{cases} -\sqrt{r_i A_i} \sqrt{\frac{r_j}{A_j}} F_{ij} & \text{if } i \neq j \\ 1 & \text{otherwise.} \end{cases}$$

The values of the scaling parameters $\sqrt{r_i A_i}$ and $\sqrt{r_j/A_j}$ depend only on the area and reflectivity values of the patches and do not change throughout the iterations. Therefore,

the values of the scaling parameters can be computed once at the beginning of the solution phase and maintained in two vectors (for each color band) representing two diagonal matrices $\mathbf{D}_1 = \text{diag}[\sqrt{r_1 A_1}, \dots, \sqrt{r_N A_N}]$ and $\mathbf{D}_2 = \text{diag}[\sqrt{r_1/A_1}, \dots, \sqrt{r_N/A_N}]$. The basic steps of the Scaled Conjugate-Gradient algorithm (**SCG**) proposed for the solution phase of the radiosity method is illustrated in Fig. 2.2. The \mathbf{p}^k and $\tilde{\mathbf{r}}^k$ vectors in Fig. 2.2 denote the direction and residual vectors at iteration k , respectively. Note that, $\tilde{\mathbf{r}}^k = \tilde{\mathbf{e}} - \tilde{\mathbf{S}}\tilde{\mathbf{b}}^k$ must be null when $\tilde{\mathbf{b}}^k$ is coincident with the solution vector.

The matrix-vector product $\mathbf{q}^k = \tilde{\mathbf{S}}\mathbf{p}^k$ looks as if the $\tilde{\mathbf{S}}$ matrix is to be computed and stored for each color band. However, this matrix-vector product can be rewritten for each color as

$$\begin{aligned} \mathbf{q}^k(r, g, b) &= \tilde{\mathbf{S}}(r, g, b)\mathbf{p}^k(r, g, b) \\ &= [\mathbf{I} - \mathbf{D}_1(r, g, b)\mathbf{F}\mathbf{D}_2(r, g, b)]\mathbf{p}^k(r, g, b) \\ &= \mathbf{p}^k(r, g, b) - \mathbf{D}_1(r, g, b)\mathbf{F}\mathbf{D}_2(r, g, b)\mathbf{p}^k(r, g, b). \end{aligned} \quad (2.10)$$

Hence, it suffices to compute and store only the original \mathbf{F} matrix, and two scaling vectors \mathbf{D}_1 and \mathbf{D}_2 for each color band for the **SCG** method. However, in order to minimize the computational overhead during the iterations due to this storage scheme, the vector $\mathbf{D}_1\mathbf{F}\mathbf{D}_2\mathbf{p}^k$ should be computed as a sequence of three matrix-vector products, $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$, $\mathbf{y} = \mathbf{F}\mathbf{x}$ and $\mathbf{z} = \mathbf{D}_1\mathbf{y}$ which take $\Theta(N)$, $\Theta(M)$ and $\Theta(N)$ times respectively. Since $M = O(N^2)$, the computational overhead due to the diagonal-matrix-vector products $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$, $\mathbf{z} = \mathbf{D}_1\mathbf{y}$ and the vector subtraction $\mathbf{q}^k = \mathbf{p}^k - \mathbf{z}$ (which also takes $\Theta(N)$ time) is negligible. The computational complexity of a single **SCG** iteration is

$$T_{SCG} \approx (2M + 18N)t_{calc}. \quad (2.11)$$

Although the above operations convert the \mathbf{C} matrix into a symmetric matrix, in practice one should be careful when using the **SCG** method. The hemi-cube method used in the form-factor calculations is an approximation. As a result, the form-factor values calculated may contain numeric errors due to violation of some assumptions [7]. Therefore, the reciprocity relation may not hold due to these numerical errors, and above operations may still result in a non-symmetric matrix.

Initially, choose $\tilde{\mathbf{b}}^0$ and let $\tilde{\mathbf{r}}^0 = \tilde{\mathbf{e}} - \tilde{\mathbf{S}}\tilde{\mathbf{b}}^0$ and then compute $\langle \tilde{\mathbf{r}}^0, \tilde{\mathbf{r}}^0 \rangle$

for $k = 0, 1, 2, \dots$

1. form $\mathbf{q}^k = \tilde{\mathbf{S}}\mathbf{p}^k$ as
 $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$; $\mathbf{y} = \mathbf{F}\mathbf{x}$; $\mathbf{z} = \mathbf{D}_1\mathbf{y}$; $\mathbf{q}^k = \mathbf{p}^k - \mathbf{z}$
 2. (a) $\theta = \langle \mathbf{p}^k, \mathbf{q}^k \rangle$
 (b) $\alpha = \frac{\langle \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k \rangle}{\theta}$
 3. $\tilde{\mathbf{r}}^{k+1} = \tilde{\mathbf{r}}^k - \alpha\mathbf{q}^k$
 4. $\tilde{\mathbf{b}}^{k+1} = \tilde{\mathbf{b}}^k + \alpha\mathbf{p}^k$
 5. (a) $\gamma = \langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle$
 (b) $\beta = \frac{\gamma}{\langle \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k \rangle}$
 (c) $\mathbf{r}^k \leftarrow \mathbf{D}_2\tilde{\mathbf{r}}^k$, $\mathbf{b}^k \leftarrow \mathbf{D}_2\tilde{\mathbf{b}}^k$
 check $\text{Norm}(\mathbf{r}^k)/\max(\mathbf{b}^k) < \epsilon$
 (d) $\langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle = \gamma$
 6. $\mathbf{p}^{k+1} = \tilde{\mathbf{r}}^{k+1} + \beta\mathbf{p}^k$
-

Figure 2.2: Basic steps of SCG method.

Convergence Check

The convergence of iterative methods is usually checked by comparing a selected norm of the residual error vector $\mathbf{r}^k = \mathbf{e} - \mathbf{C}\mathbf{b}^k$ with a predetermined threshold value at each iteration k . In this work, the following error norm is used for the convergence check

$$\text{error}^k = \frac{\sum_{i=1}^N |r_i^k|}{\max(|b_i^k|)} \quad (2.12)$$

where $|\cdot|$ denotes the absolute value. Iterations are terminated when error becomes less than a predetermined threshold value (e.g., $\text{error}^k < \epsilon$ where $\epsilon = 5 \times 10^{-6}$). Note that, the residual vector \mathbf{r}^k is already computed in the SCG method. On the other hand, the residual vector $\mathbf{r}^k = \mathbf{e} - \mathbf{C}\mathbf{b}^k$ is not explicitly computed in the GJ scheme. However, note that,

$$\mathbf{r}^k = \mathbf{e} - \mathbf{C}\mathbf{b}^k = \mathbf{e} - (\mathbf{I} - \mathbf{R}\mathbf{F})\mathbf{b}^k = \mathbf{e} + \mathbf{R}\mathbf{F}\mathbf{b}^k - \mathbf{b}^k$$

$$= \mathbf{b}^{k+1} - \mathbf{b}^k. \quad (2.13)$$

Hence, the residual error vector \mathbf{r}^k can easily be calculated at each iteration of the **GJ** scheme by a single vector subtraction operation.

2.2 Previous Work on Parallel Gathering Radiosity

There are various parallel implementations for progressive refinement and gathering methods in the literature [12, 13, 73, 74, 14, 27, 35, 44, 24, 94, 67, 6, 22]. In this section, parallel approaches for gathering method are summarized.

One of the approaches is by Price and Truman [73]. In their work, the gathering method was parallelized on a transputer based architecture where processors organized as a ring having a master processor, used for communicating with host and graphics system, and a number of slave processors to do the calculations. Any data exchange can be done using this ring interconnection. In their approach, they assume that total scene data can be replicated in the local memories of the processors, hence form-factor computations can be done without any inter-processor communication. The **GJ** iterative scheme is used in the solution phase.

Another approach is by Purgathofer and Zeiller [74]. In their approach, they used a ring of transputers. In form-factor computation phase, “receiving” patches are statically distributed to worker processors. The distribution of patches to processors are done randomly to obtain a better load balance. The master processor sends global patch information in blocks to first processor in the ring. Then, the patch information is circulated in the ring. In their approach, the sparsity of form-factor matrix is exploited and matrix is maintained in compressed form. The memory used for matrix rows and hemi-cube information is overlapped allowing calculation of several rows of matrix at a time in each processor. The number of rows calculated at a time decreases as more rows allocate the memory shared with hemi-cube information.

Chalmers and Paddon [12, 13] use demand-driven approach in the form-factor computation phase and data-driven approach in the solution phase where data is assigned to processors in a static manner. The target architecture is based on transputers arranged

in minimal path length structure. They discuss the trade-offs between demand and data-driven schemes in the parallelization of the form-factor computation phase in [67]. In the former work [12], they addressed the need for data re-distribution for better load balancing in the solution phase. In their work in [13], a demand-driven approach is used for form-factor calculation phase. In that work, the form-factor row computations are conceptually divided evenly among the processors. The even decomposition here refers to the equal number of row allocation to each conceptual region. Each processor is assigned a task by the master from its conceptual region until all tasks in its region are consumed. Idle processors whose conceptual regions are totally consumed are assigned tasks from the conceptual regions of other processors. However, in such cases, the computed form-factor vectors are passed to the processors which own the conceptual region. The **GJ** iterative scheme is used in the solution phases of all these works.

2.3 Intel's iPSC/2 Hypercube Multicomputer

Intel iPSC/2 hypercube multicomputer is a distributed-memory multicomputer. Processors can perform different operations on different data simultaneously. There is no shared memory in the system. Data exchange between processors and synchronization of processors is done via exchanging messages between processors. A brief description of hypercube interconnection topology [79, 75] and Intel's iPSC/2 hypercube multicomputer is given here.

A d -dimensional hypercube consists of $P = 2^d$ processors (nodes) with a link between every pair of processors whose binary addresses differ in one bit. Thus, each processor is connected to d other processors. The *hamming-distance* between two processors in a hypercube is defined to be the number of different bits between these two processors' ids. *Channel i* refers to the communication link between processors whose processor ids differ in only i^{th} bit. A 4-dimensional hypercube with binary encoding of the nodes is illustrated in Fig. 2.3. The circles in the figure represents a memory and a processor pair.

Many other topologies, such as ring and mesh, can be embedded onto hypercube

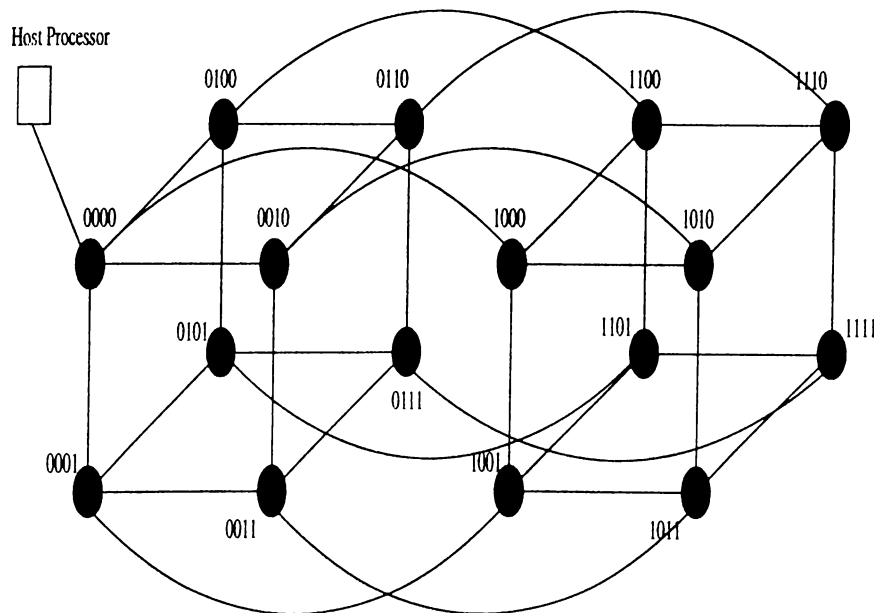
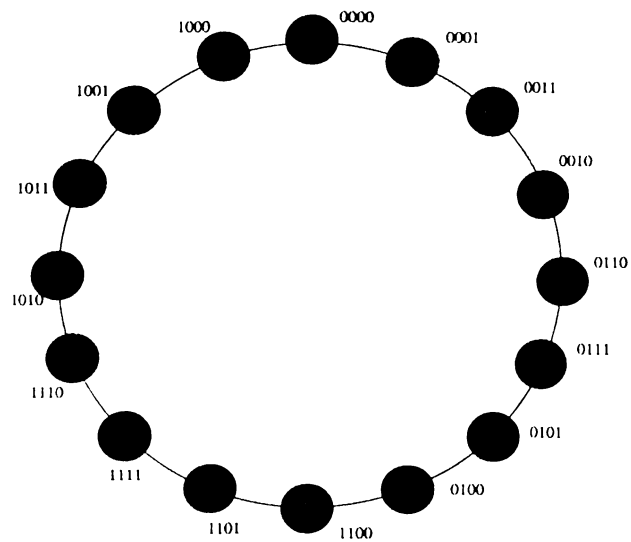


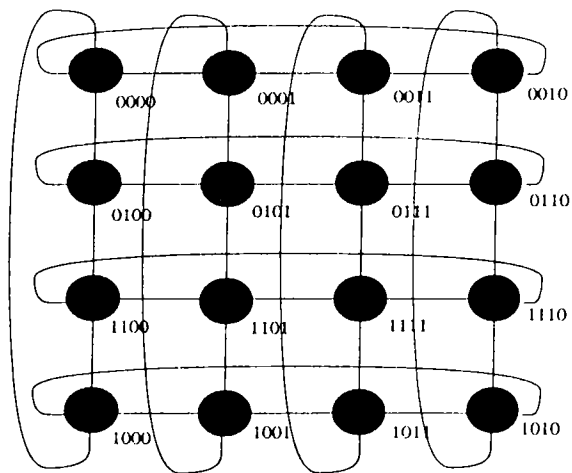
Figure 2.3: A 16 node hypercube multicomputer

topology. Therefore, it is possible to arrange processors to the most suitable interconnection topology for the solution of the problem. A ring embedding and a 2-dimensional mesh embedding are given in Fig. 2.4 (a) and (b), respectively.

Each node processor of the iPSC/2 has 4 MBytes of local memory and 16-MHz 80386 CPU with 80387 math co-processor. The 80386/80387 pair can achieve a peak performance of $300K Flops$. Any communication on hypercube multicomputer is done via using hardware modules called *Direct Connect Modules* (DCMs). Any node in the hypercube can communicate to any other node even if they are not connected directly. Communication between such nodes are done using the *DCMs* of other processors. Since communications are handled by these hardware units, processors on the communication route are not interrupted. The setup time (t_{su}) of Intel's iPSC/2 hypercube is about 400 microseconds, and a maximum of ≈ 2.8 MBytes can be transmitted in one second from one processor to a neighbor processor. A host processor with 8 MBytes of memory and with a 80386/80387 pair is also available to interface the nodes to disk and to user. This host processor can sometimes be used for managing and synchronizing the nodes of the hypercube. The host processor is connected to node 0 by a communication link.



(a)



(b)

Figure 2.4: (a) A ring embedding (b) A 2-dimensional mesh embedding into a 4-dimensional hypercube.

2.4 Parallel Computation of the Form-Factor Matrix

In this section, parallel algorithms devised for form-factor matrix computation phase of the gathering method are described. The parallel implementation of this phase requires the decomposition and mapping of data and calculations among the processors of the hypercube.

2.4.1 Static Assignment

In this scheme, each processor is statically assigned the responsibility of computing the rows corresponding to a subset of patches prior to the parallel execution of this phase. However, projection computations onto local hemi-cubes may introduce load imbalance during the parallel form-factor computation phase. The complexity of the projection of an individual patch onto a hemi-cube depends on several geometric factors. A patch which is clipped completely requires much less computation compared to a visible patch since it leaves the projection pipeline in a very early stage. Furthermore, a patch with larger projection area on a hemi-cube requires more scan-conversion computation than a patch with a smaller projection area. Therefore, the assignment scheme should be carefully selected in order to maintain the load balance in this phase.

In this work, we recommend two types of static assignment schemes, *scattered* and *random*. In the scattered assignment scheme, the adjacent patches on each surface should be ordered consecutively. Then, the successive patches in the sequence are assigned to the processors in a round-robin fashion. That is, the first patch is assigned to processor 0, the next to processor 1, etc. When P patches are assigned, the next patch is assigned to processor 0 and this process continues. Here, P denotes the number of processors in the hypercube. Note that, hemi-cube fill process for the adjacent patches is expected to take almost equal amount of computation due to the similar view-volume of adjacent patches. Hence, scattered assignment is expected to yield good load balance. The scattered assignment of the patches on a regular surface (e.g. rectangular surface) is trivial. Unfortunately, this assignment scheme may necessitate expensive preprocessing computations for the irregular surfaces. The random assignment scheme is recommended if the scene data is not suitable for the preprocessing needed for the scattered assignment

scheme. In this assignment scheme, randomly selected patches are similarly assigned to the processor in a round-robin fashion. It is experimentally observed that random assignment scheme yields fairly good load balance for sufficiently large $\frac{N}{P}$ ratio. The random assignment scheme is used in this work.

In both of these two assignment schemes, first $(N \bmod P)$ processors in the decimal processor ordering are assigned $\lceil \frac{N}{P} \rceil$ patches whereas the remaining processors are assigned $\lfloor \frac{N}{P} \rfloor$ patches. After the assignment, patches are re-numbered so that $\frac{N}{P}$ patches assigned to processor ℓ are re-numbered from $\frac{N}{P}\ell$ to $\frac{N}{P}(\ell + 1) - 1$. The new global numbering (new patch-id's) is not modified throughout the computations.

Patch Circulation

In this scheme, the host processor distributes only local patch information to node processors. After receiving the local patch information, each processor calculates the rows of the \mathbf{F} matrix for its local patches. Each processor places a hemi-cube around the center of a local patch and calculates the form-factor row for that patch. The calculation of a form-factor row requires the projection of all patches in the scene to the respective hemi-cube. Each processor's local patch information is circulated among the processors so that each processor can project all patches to their local hemi-cubes. The ring embedded hypercube structure, which can easily be achieved by *gray code* ordering [75], is used for patch circulation. In the first concurrent step of the circulation, processors send a copy of their local patch data to their left processor on the ring after projecting this local set of patches onto their current local hemi-cube. In the following concurrent steps, processors project the set of patches received from their right neighbor onto their current hemi-cube and then send this set of patches to their left neighbor. If the number of patches N is not a multiple of P , those processors having $\lceil \frac{N}{P} \rceil$ local patches require one more patch circulation phase than the processors having $\lfloor \frac{N}{P} \rfloor$ local patches. Hence, those processors having $\lfloor \frac{N}{P} \rfloor$ patches participate in an extra patch circulation phase, which does not include any local hemi-cube fill operation, for the sake of other processors. The node algorithm in pseudo-code for this scheme is given in Fig. 2.5. It is invoked with `Set_of_patches = Set_of_all_local_patches` and `MaxIter = $\lceil \frac{N}{P} \rceil$` .

```

procedure PATCH_CIRCULATION(Set_of_patches.MaxIter)
  for each patch "p" in Set_of_patches do
    Place a hemi-cube around the patch "p";
    Project patches in Set_of_all_local_patches onto the hemi-cube;
    /* data circulation in ring */
    Current_patch_data = Set_of_all_local_patches;
    repeat P-1 times do
      Send Current_patch_data to the left neighbor processor;
      Receive patch data from right neighbor processor;
      Project the received patches onto the hemi-cube;
      Current_patch_data = Received_patch_data;
    Compute the corresponding row of the local form-factor matrix;
  IterStep = Num_of_patches in the Set_of_patches;
  while ( IterStep < MaxIter ) do
    /* data circulation in ring */
    Current_patch_data = Set_of_all_local_patches;
    repeat P-1 times do
      Send Current_patch_data to the left neighbor processor;
      Receive patch data from right neighbor;
      Current_patch_data = Received_patch_data;
    IterStep = IterStep + 1;

```

Figure 2.5: The node algorithm in pseudo-code for patch circulation scheme.

Assuming a perfect computational load balance, information for $\Theta(\frac{N}{P})$ patches is concurrently transmitted between successive processors of the ring in each communication step. The total concurrent communication volume in a single circulation step is then $\Theta(\frac{N}{P})(P-1) = \Theta(N)$. Hence, the total concurrent communication volume is $\Theta(N)[\frac{N}{P}] = \Theta(\frac{N^2}{P})$. This communication overhead can be reduced by avoiding communication as much as possible by duplicating the global patch information at each node processor. The scheme to implement this idea is given in the following section.

```

procedure AVOIDING_COMMUNICATION
  for each local patch do
    Place a hemi-cube around the local patch;
    Project global patches onto the hemi-cube.
    Calculate corresponding row of the local portion
    of form-factor matrix;
    if ( memory is full ) then
      Broadcast MEMORY_FULL message to other nodes;
      Calculate the number of remaining rows;
      Terminate the for loop;
    if ( MEMORY_FULL message is received ) then
      Calculate the number of remaining rows;
      Terminate the for loop;
  Free the space allocated to non-local patch information;
  Perform global maximum operation to obtain the
  maximum (max_remaining) of the number of remaining rows;
  /* if there are rows, not finished yet */
  if ( max_remaining not equal to 0 ) then
    call PATCH_CIRCULATION(Set_of_remaining_local_patches,max_remaining);

```

Figure 2.6: The node algorithm in pseudo-code for form-factor computation by storage sharing scheme.

Storage Sharing Scheme

Using dynamic memory allocation for the computed \mathbf{F} matrix rows can be exploited to share the memory needed for global patch information with the memory to be allocated for non-zero matrix elements. With such a sharing of memory, we can avoid inter-processor communication until the memory allocated to global patch information is required for a row of the matrix.

In this scheme, the global patch information is duplicated in each processor after the local patch assignment and the corresponding global patch re-numbering mentioned earlier. Then, processors concurrently compute and store the form-factor rows corresponding to their local assignment avoiding inter-processor communication until no more memory can be allocated for the new row. The node algorithm in pseudo-code

for this method is given in Fig. 2.6. If a processor cannot allocate memory space for storing the computed from-factor row, it broadcasts `MEMORY_FULL` message so that other processors can switch to communication phase as soon as possible and run *patch circulation* scheme. Each processor, which receives the message or which cannot allocate memory, calculates the number of remaining local rows to be computed. Before starting to patch circulation, the space allocated to global patch information is deallocated (only space allocated to local patches is not freed) to obtain space for the remaining rows. Note that the patch circulation should be performed until all remaining rows of the **F** matrix are calculated. Therefore, the data circulation phase in *patch circulation* scheme (Fig. 2.5) should be repeated by the number of times equal to the maximum of the number of unprocessed patches remaining. This number can be found by performing a global maximum operation on the number of remaining patches in each processor after the **for** loop (Fig. 2.6) is terminated. The global maximum operation requires $\log_2 P$ concurrent exchange communication steps and its communication structure is the same as that of the global concatenate operation illustrated in Fig. 2.8.

2.4.2 Demand-Driven Assignment Scheme

This approach is an attempt to achieve better load balance through patch assignment to idle processors upon request. The scheme proposed in this work differs from the approach used in [12, 13]. Unlike their scheme, no *conceptual* division of patches is done. When a patch is processed by a processor, the computed form-factor row remains in that processor. In this scheme, each node processor demands a new patch assignment from the host processor as soon as it computes the form-factor row(s) associated with the previous patch assignment. Host processor sends the necessary information for a predefined number of patch assignments to the requesting node processor. The number of patch assignments at a time is a factor affecting the performance. The number of node-to-host and host-to-node communications decrease with increasing number of patch assignments at a time. However, this may decrease the quality of load balance.

Each node processor keeps an array to save the reflectivity and emission values of the processed patches. The global ids of the processed patches are also saved in an array to be used in the solution phase. In addition, each processor holds the information for all

patches in the scene to avoid inter-processor communication as is explained in storage sharing scheme. Host processor behaves as a master. It is responsible for supplying demands and synchronizing nodes between different phases. The host program maintains an array for global patch information and keeps account of the remaining patches to be processed. All node processors are synchronized by the host processor when one or more of the nodes' memory becomes full and processors have to switch to patch circulation mode. Host is also responsible for the termination of form-factor computation phase.

2.5 Parallel Solution Phase

This section describes the parallel **GJ** and **SCG** algorithms developed for the solution phase. The parallel implementation of the solution phase is closely related to the schemes used in the form-factor matrix computation phase because patch distribution, thus row distribution, to processors differs in each scheme. In the following sections, parallel algorithms for the solution phase are described assuming static assignment scheme is used in the parallel form-factor computation phase. An efficient parallel renumbering scheme is described in Section 2.5.3 to adapt these algorithms if demand-driven assignment scheme is used in the parallel form-factor computation phase.

2.5.1 Parallel Gauss-Jacobi Method

The **GJ** algorithm formulated (Fig. 2.1) for the solution phase has the following basic types of operations: matrix-vector product ($\mathbf{x} = \mathbf{F}\mathbf{b}^k$), diagonal-matrix vector product ($\mathbf{y} = \mathbf{R}\mathbf{x}$), vector subtraction/addition operations ($\mathbf{b}^{k+1} = \mathbf{y} + \mathbf{e}$, $\mathbf{r}^k = \mathbf{b}^{k+1} - \mathbf{b}^k$), vector norm and maximum operations (step 3). All of these basic operations can be performed concurrently by distributing the rows of the form-factor matrix \mathbf{F} , and the corresponding diagonals of the \mathbf{R} matrix and the corresponding entries of the \mathbf{b} and \mathbf{e} vectors. In the parallel form-factor computation phase, each processor computes the complete row of form-factors for its local patches. Hence, each processor holds a row slice of the form-factor matrix at the end of the form-factor computation phase. Thus, the row partitioning required for the parallelization of the solution phase is automatically achieved in the form-factor computation phase. The slices of the \mathbf{R} , \mathbf{b} and \mathbf{e} vectors are

Initially, choose \mathbf{b}^0
 for $k = 1, 2, 3, \dots$

1. (a) perform global-concatenate on $\mathbf{b}_{local}^k \rightarrow \mathbf{b}_{global}^k$
 (b) $\mathbf{x}_{local} = \mathbf{F}_{local} \mathbf{b}_{global}^k$
 (c) $\mathbf{y}_{local} = \mathbf{R}_{local} \mathbf{x}_{local}$
 (d) $\mathbf{b}_{local}^{k+1} = \mathbf{y}_{local} + \mathbf{e}_{local}$
 2. $\mathbf{r}_{local}^k = \mathbf{b}_{local}^{k+1} - \mathbf{b}_{local}^k$
 3. (a) $\sigma_{local} = \text{Norm}(\mathbf{r}_{local}^k)$
 $b_{localmax} = \max(\mathbf{b}_{local}^k)$
 (b) perform one global-sum-max operation to compute
 $\sigma_{local} \rightarrow \sigma_{global}$ and $b_{localmax} \rightarrow b_{globalmax}$
 check $\sigma_{global}/b_{globalmax} < \epsilon$
-

Figure 2.7: Parallel **GJ** algorithm.

mapped to the processors accordingly. Each processor is responsible for updating its own slice of the global \mathbf{b} vector in a local B array (of size N/P) at each iteration. Figure 2.7 illustrates the parallel **GJ** algorithm. In an individual **GJ** iteration, each processor needs to perform a local matrix-vector product which involves $\frac{N}{P}$ inner products of its local rows with the global \mathbf{b} vector. In order to perform this matrix-vector product, the whole \mathbf{b} vector computed in a distributed manner in a particular iteration is needed by all processors in the next iteration. This requirement necessitates the *global concatenate* operation which is illustrated for a 3-dimensional hypercube topology in Fig. 2.8. In the global concatenate operation each processor ℓ moves its local \mathbf{b} array to the ℓ^{th} slice of a working array GB of size N . Then, $\log_2 P$ concurrent exchange communication steps are performed between neighbor processors over channels $j = 0, 1, 2, \dots, \log_2 P - 1$ as is illustrated in Fig. 2.8. Note that the amount of concurrent data exchange between processors is only $\frac{N}{P}$ in the first step and it is doubled at each successive step. That is, at the j^{th} communication step, processors exchange the appropriate slices of size $2^{(j-1)} \frac{N}{P}$ of their local GB array over channel $j - 1$. Therefore, the total volume of concurrent

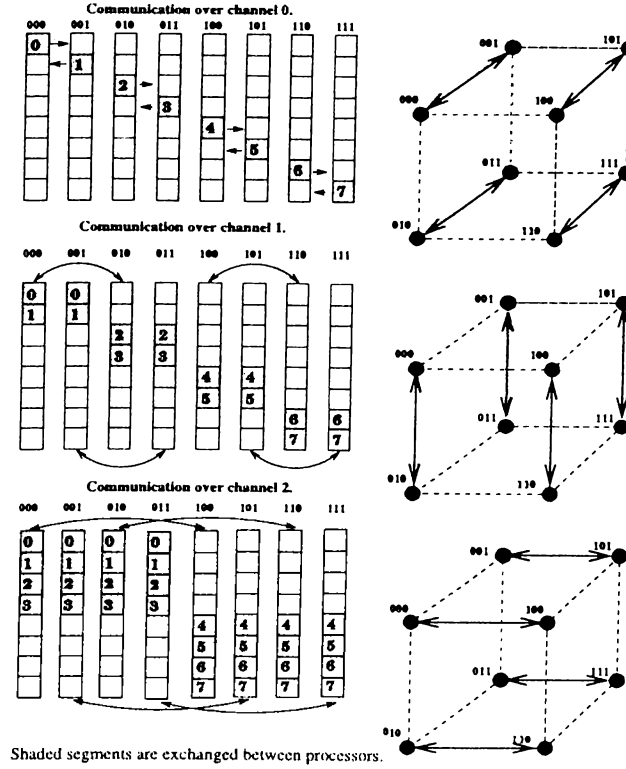


Figure 2.8: Global concatenate operation for a 3-dimensional hypercube.

communication is

$$Volume\ of\ communication = \sum_{j=0}^{\log_2 P - 1} \frac{2^j N}{P} = \frac{(P-1)N}{P} \quad words. \quad (2.14)$$

The distributed vector add/subtract operations are performed concurrently as local vector operations on vectors of size $\frac{N}{P}$ without necessitating any interprocessor communication. The partial sums computed by each processor must be added to form the global sum to compute the vector norm $(\sum_{i=1}^N |r_i|)$. In addition, local $b_{localmax}$ values should be compared to obtain the global maximum ($b_{globalmax}$). Furthermore, the results should be distributed to all processors in order to ensure the termination at the same iteration. The distributed global norm operation can be performed by *global-sum* operation. The exchange-communication sequence of the global-sum operation is exactly the same as that of global-concatenate operation. The only difference is the local scalar addition operation performed after each exchange communication step [4] instead of the local vector

concatenate operation. This local scalar operation involves the addition of the received partial-sum to the current partial-sum. Similarly, distributed global maximum can be found by using *global-max* operation. The global-max operation can be done by replacing the local scalar addition operation with comparison operation in global-sum operation. Performing global-sum and global-max operations successively requires $2\log_2 P$ set-up time. Fortunately, this set-up time can be decreased to $\log_2 P$ by combining global-max and global-sum operations into a single global operation (global-sum-max). In this global operation, partial sums and current maximums are exchanged after each exchange communication step. Therefore, assuming perfect load balance, the parallel computational complexity of an individual **GJ** iteration is

$$T_{GJ} \approx \left(\frac{2M}{P} + \frac{6N}{P}\right)t_{calc} + 2\log_2 P t_{su} + \left(\frac{P-1}{P}N + 2\log_2 P\right)t_{tr}. \quad (2.15)$$

As is seen from this equation, communication overhead can be considered negligible for sufficiently large granularity (i.e. $M/P \gg N$). Note that this equation is equivalent to Eq. (2.7) for $P = 1$.

2.5.2 Parallel Scaled Conjugate-Gradient Method

The **SCG** algorithm formulated (Fig. 2.2) for the solution phase has the following basic types of operations: matrix-vector product ($\mathbf{y} = \mathbf{F}\mathbf{x}$), diagonal matrix-vector products ($\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$, $\mathbf{z} = \mathbf{D}_1\mathbf{y}$), vector subtraction ($\mathbf{q}^k = \mathbf{p}^k - \mathbf{z}$), inner-products ($\langle \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k \rangle$, $\langle \mathbf{p}^k, \mathbf{q}^k \rangle$), vector update operations in steps 3, 4, and 6, and vector norm and maximum operations for the convergence check. All of these basic operations can be performed concurrently by distributing the rows of the **F** matrix and the corresponding entries of the **e**, **D**₁, **D**₂, **x**, **y**, **z**, **b**, **p**, **r̃**, and **q** vectors.

As discussed in parallel **GJ** algorithm, during parallel form-factor computation phase, the rows of the **F** matrix is assigned to the processors automatically. With such a mapping each processor stores its own (local) row slice of the **F** matrix and the corresponding slices of the **e**, **D**₁, and **D**₂ vectors. Furthermore, each processor is responsible for updating its local slices of the **x**, **y**, **z**, **b**, **p**, **r̃**, **q** vectors. Figure 2.9 illustrates the parallel **SCG** algorithm.

for $k = 0, 1, 2, \dots$

1. (a) $\mathbf{x}_{local} = \mathbf{D}_{2(local)} \mathbf{p}_{local}^k$
 (b) perform global-concatenate on $\mathbf{x}_{local} \rightarrow \mathbf{x}_{global}$
 (c) $\mathbf{y}_{local} = \mathbf{F}_{local} \mathbf{x}_{global}$
 (d) $\mathbf{z}_{local} = \mathbf{D}_{1(local)} \mathbf{y}_{local}$
 (e) $\mathbf{q}_{local}^k = \mathbf{p}_{local}^k - \mathbf{z}_{local}$
 2. (a) $\theta_{local} = \langle \mathbf{p}_{local}^k, \mathbf{q}_{local}^k \rangle$
 (b) perform global-sum on $\theta_{local} \rightarrow \theta_{global}$
 (c) $\alpha_{global} = \langle \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k \rangle_{global} / \theta_{global}$
 3. $\tilde{\mathbf{r}}_{local}^{k+1} = \tilde{\mathbf{r}}_{local}^k - \alpha_{global} \mathbf{q}_{local}^k$
 4. $\tilde{\mathbf{b}}_{local}^{k+1} = \tilde{\mathbf{b}}_{local}^k + \alpha_{global} \mathbf{p}_{local}^k$
 5. (a) $\lambda_{local} = \langle \tilde{\mathbf{r}}_{local}^{k+1}, \tilde{\mathbf{r}}_{local}^{k+1} \rangle$
 (b) $\mathbf{r}_{local}^k \leftarrow \mathbf{D}_2 \tilde{\mathbf{r}}_{local}^k, \mathbf{b}_{local}^k \leftarrow \mathbf{D}_2 \tilde{\mathbf{b}}_{local}^k$
 $b_{localmax} = \max(\mathbf{b}_{local}^k), \sigma_{local} = \text{Norm}(\mathbf{r}_{local}^k)$
 (c) perform one global operation to compute
 $\lambda_{local} \rightarrow \lambda_{global}, \sigma_{local} \rightarrow \sigma_{global}, b_{localmax} \rightarrow b_{globalmax}$
 check $\sigma_{global} / b_{globalmax} < \epsilon$
 (d) $\beta_{global} = \lambda_{global} / \langle \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k \rangle_{global}$
 (e) $\langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle_{global} = \lambda_{global}$
 6. $\mathbf{p}_{local}^{k+1} = \tilde{\mathbf{r}}_{local}^{k+1} + \beta_{global} \mathbf{p}_{local}^k$
-

Figure 2.9: Parallel SCG method.

A sequence of distributed matrix and vector computations are needed for the distributed computation of the \mathbf{q}^k vector. To perform the diagonal matrix-vector products $\mathbf{x} = \mathbf{D}_2 \mathbf{p}^k$ and $\mathbf{z} = \mathbf{D}_1 \mathbf{y}$, processors concurrently compute their local \mathbf{x}_{local} and \mathbf{z}_{local} vectors by performing element-by-element product of pairs of local vectors which correspond to their slices of the global $\mathbf{D}_2, \mathbf{p}^k$ and \mathbf{D}_1, \mathbf{y} vectors, respectively. Thus, distributed diagonal matrix-vector product does not necessitate any interprocessor communication. As is also the case in the **GJ** method, the distributed computation of the matrix-vector product $\mathbf{y} = \mathbf{F} \mathbf{x}$ necessitates global-concatenate operation on the local \mathbf{x}_{local} vector stored in a local array X of size N/P in each processor. Then, processors concurrently compute their local \mathbf{y}_{local} vectors by multiplying a local matrix corresponding to their slice of

the \mathbf{F} matrix with the global \mathbf{x}_{global} vector, collected in an array $G\mathbf{X}$ of size N in their local memories after the global-concatenate operation. Finally, processors concurrently compute their local \mathbf{q}_{local}^k vectors by performing local vector subtraction operations.

All processors need the most recently updated values for the global scalars α_{global} and β_{global} in order to perform their local vector updates in steps 3, 4, and 6. As is seen in steps 2 and 5, the update of these global scalars involve the computation of the inner-products $\langle \mathbf{q}^k, \mathbf{p}^k \rangle$ and $\langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle$ at each iteration. Hence, all processors should receive the results of these distributed inner-product computations. In order to perform these distributed inner-products, processors concurrently compute the local inner-products (partial sums) corresponding to their slices of the respective global vectors. Then, the inner-product result is accumulated in the local memory of each processor by performing a global-sum operation. At the end of the global-sum operation, processors can concurrently compute the same value for the global scalars α_{global} and β_{global} using these global inner-product results. In steps 3, 4, and 6, processors concurrently update their local \mathbf{b}_{local} , $\tilde{\mathbf{r}}_{local}$ and \mathbf{p}_{local} vectors. Note that the distributed norm and maximum operations also necessitate global-sum-max operation after all processors concurrently compute their local (partial) error norms and maximums which correspond to the norm/maximum of their slices of the global \mathbf{r} and \mathbf{b} vectors. Fortunately, these operations and the global inner-product $\langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle$ can be concurrently accumulated in the same global operation to avoid the extra $\log_2 P$ set-up time overhead. In this global operation, one local maximum and two partial sums are exchanged in each step of the $\log_2 P$ concurrent exchange steps. Therefore, assuming perfect load balance, the parallel computational complexity of an individual SCG iteration is

$$T_{SCG} \approx \left(\frac{2M}{P} + \frac{18N}{P} \right) t_{calc} + 3 \log_2 P t_{su} + \left(\frac{P-1}{P} N + 4 \log_2 P \right) t_{tr}. \quad (2.16)$$

As is seen from this equation, communication overhead can be considered negligible for sufficiently large granularity (i.e. $M/P \gg N$).

2.5.3 A Parallel Renumbering Scheme

In the form-factor computation phase in static assignment scheme, patches are renumbered and assigned to the node processors such that processor ℓ has patches from $\frac{N}{P}\ell$ to

$\frac{N}{P}(\ell + 1) - 1$ for $\ell = 0, 1, 2, \dots, P - 1$. Therefore, the exchange sequence together with the local concatenate scheme in the global-concatenate operations at step 1(a) of parallel **GJ** algorithm and step 1(b) of parallel **SCG** algorithm maintains the original global patch ordering of the static assignment scheme in the local copies of the global vectors. During the concurrent local matrix vector products, the appropriate entries in the current global vectors collected in local *GB* (in **GJ**) and *GX* (in **SCG**) arrays can be accessed for multiplication by indexing through the *column-ids* of the local non-zero form-factor values. Unfortunately, this nice consistency between the global patch numbering and the global **F** matrix row ordering among the processors is disturbed in the demand-driven assignment scheme. So, the demand-driven assignment scheme necessitates two level indexing (indirections) for each scalar multiplication involved in the local matrix-vector products at each iteration. We propose an efficient parallel renumbering scheme to avoid this two-level indexing.

During the form-factor matrix computation phase of the demand-driven assignment scheme each processor saves the global-id of the patches it receives from the host processor in a local integer array *ID*. After the form-factor computation phase, a global-concatenate operation on these local arrays is performed to collect a copy of global *GID* array in each processor. Note that the collection operation is done in the same way as the global-concatenate operation to be performed on the local *B* (in **GJ**) and *X* (in **SCG**) arrays during the iterations. Hence, there is a one-to-one correspondence between the *GB* (*GX*) and *GID* arrays such that the radiosity value in *GB*[*i*] (*GX*[*i*]) belongs to the patch whose original global-id=*GID*[*i*]. Then, each processor constructs the same permutation array *PERM* of size *N*, where *PERM*[*GID*[*i*]] = *i* (for *i*=1,2,...,*N*) by performing a single for-loop. Here, *PERM*[*i*] denotes the new global-id for the *i*th patch in the original global numbering. Then, each processor concurrently updates the *column_id* values of all its local non-zero form-factor values using *PERM* array as *column_id* = *PERM*[*column_id*]. Note that this renumbering operation is performed only once as a preprocessing step just before the solution phase, and it is not repeated when the reflectivity and/or emission values are modified.

2.6 Load Balancing in the Solution Phase: Data Redistribution

Assigning equal number of rows of matrix \mathbf{F} and the corresponding vector elements suffices to achieve perfect load balance during the distributed vector operations involved in the \mathbf{GJ} and \mathbf{SCG} iterations. However, the computational complexities of individual \mathbf{GJ} and \mathbf{SCG} iterations are bounded by the distributed matrix-vector products \mathbf{Fb} and \mathbf{Fx} , respectively. So, the load balance during the distributed matrix-vector products is much more crucial than that of the vector operations. Since we exploit the sparsity of the \mathbf{F} matrix in the matrix-vector products, the load balance in these computations can only be achieved by assigning equal number of non-zero entries of the \mathbf{F} matrix to processors.

The factors that effect the load balance in the form-factor computation and the solution phases are not the same. The assignment schemes mentioned earlier for the form-factor computation phase aim at achieving load balance on the hemi-cube fill operations associated with the patches. However, even if two patches require almost equal time for the hemi-cube fill operation, the number of non-zero entries in the respective rows may be substantially different. Thus, an assignment scheme (e.g., demand-driven assignment) which yields near perfect load balance in the form-factor computation phase may not achieve a good load balance during the solution phase. Furthermore, it is not possible to achieve perfect load balance in the form-factor computation phase through static assignment since the amount of projection work is not known a priori. However, once the sparsity structure of the \mathbf{F} matrix is determined at the end of the form-factor computation phase, static re-assignment can be utilized for better load balancing in the solution phase. In other words, a redistribution of \mathbf{F} matrix entries is needed for better load balance during the distributed matrix-vector product operations. There are various data redistribution schemes in the literature [43, 78]. The main objective in those schemes is to achieve a data redistribution in such a way that the number of data elements in different processors differ at most by one. However, these schemes do not assume any hierarchy among the data elements. In our case, data elements belong to the rows of the \mathbf{F} matrix and it is desirable to minimize the subdivision of rows among

the processors because subdivided rows may require extra communication during the solution phase. Furthermore, the data movement necessitated by the redistribution should be minimized to minimize the preprocessing overhead for the solution phase.

2.6.1 A Parallel Data Redistribution Scheme

In this section, an efficient parallel redistribution scheme is proposed. This scheme allows at most one shared row between successive processors in the decimal ordering (i.e. between processors ℓ and $\ell+1$ for $\ell=0,1,\dots,P-2$). That is, each processor ℓ , except the first and the last processors (0 and $P-1$, respectively), may share at most two rows, one with processor $\ell-1$ and one with processor $\ell+1$. The processors 0 and $P-1$ may share at most one row with processors 1 and $P-2$, respectively. Recall that successive processors in the decimal ordering hold the successive row slices of the distributed \mathbf{F} matrix. We assume a similar global implicit numbering for the non-zero entries of the distributed \mathbf{F} matrix. Non-zero entries in the same row are assumed to be numbered in the storage order. Non-zero entries in the successive rows are assumed to be numbered successively. Hence, the global numbers of the non-zero entries in processor $\ell+1$ is assumed to follow those of processor ℓ .

In the parallel re-assignment phase, a global-concatenate operation is performed on the local \mathbf{F} matrix non-zero entry counts so that each processor collects a copy of the global integer *OLDMAP* array of size P . At this stage, *OLDMAP* $[\ell]$ denotes the number of non-zero entries computed and stored in processor ℓ for $\ell = 0,1,\dots,P-1$. Then, processors concurrently run the prefix-sum operation on their *OLDMAP* array. After the prefix sum operation, *OLDMAP* $[\ell-1]+1 \dots \text{OLDMAP}[\ell]$ denotes the range of non-zero entries computed and stored in processor ℓ in the assumed ordering. Note that *OLDMAP* $[P-1] = M$ yields the total number of non-zero entries in the global \mathbf{F} matrix. Then, all processors concurrently construct the same integer *NEWMAP* array of size P , where *NEWMAP* $[\ell] = \lceil \frac{M}{P} \rceil$ for $\ell=0,1,\dots,(M \bmod P)-1$ and *NEWMAP* $[\ell] = \lfloor \frac{M}{P} \rfloor$ for $\ell=(M \bmod P),\dots,P-1$. At this stage, *NEWMAP* $[\ell]$ denotes the number of non-zero entries to be stored in processor ℓ after the data redistribution. Then, processors concurrently run the prefix-sum operation on their *NEWMAP* array. Therefore, after the prefix-sum operation *NEWMAP* $[\ell-1]+1 \dots \text{NEWMAP}[\ell]$ denotes the range of \mathbf{F} matrix non-zero

entries to be stored in processor ℓ in the assumed ordering after the redistribution. Each processor, knowing the new mapping for their current local non-zero entries, can easily determine its local row sub-slices to be redistributed and their destination processor(s). Similarly each processor, knowing the old mapping for their expected mapping after the data redistribution, can easily determine the source processor(s) from which it will receive data during the redistribution and the volume of data in each receive operation. However, sending processors should append the row structure of the data transmitted in front of the messages during the data redistribution phase. Note that consecutive row data is transmitted between processors and only the first and/or last rows of the transmitted data may be partial row(s). The receiving processors store the received data in row structure according to the global row ordering by performing simple pointer operations.

At the end of the data redistribution phase, the number of non-zero entries stored by different processors may differ at most by one. Thus, perfect load balance is achieved during the distributed sparse matrix-vector product performed at each iteration of both **GJ** and **SCG** methods. However, shared rows need special attention during these distributed matrix-vector product operations. Consider a row “ i ” (in global row numbering) shared between processors ℓ and $\ell+1$. This row corresponds to the last and first (partial) local rows of processors ℓ and $\ell+1$, respectively. During the distributed matrix-vector product these two processors accumulate the partial sums which correspond to the inner-products of their local portions of the i^{th} row of the \mathbf{F} matrix with the global right-hand-side vector. These two partial sums should be added to determine the i^{th} entry of the resultant left-hand-side vector. As a result, row sharing necessitates one concurrent interprocessor communication between successive processors after each distributed matrix-vector product. In the proposed mapping, the computations associated with the vector entries corresponding to the shared rows between processors ℓ and $\ell+1$ are assigned to the processor $\ell+1$ for $\ell=0,1,\dots,P-2$. Processors concurrently send the partial inner product results corresponding to their last local row (if it is shared) to the next processor in the decimal ordering. Only a single floating-point word is transmitted in these communications. Thus, this concurrent shift-and-add scheme for handling shared rows introduces $t_{su}+t_{tr}+t_{add}$ concurrent communication and addition overhead

per iteration of both **GJ** and **SCG** algorithms.

2.6.2 Avoiding the Extra Setup Time Overhead

We propose an efficient scheme for the **GJ** method which avoids the extra setup time overhead by incorporating this extra communication into the global concatenate operation. In the proposed scheme, the global-concatenate operation is performed on \mathbf{b}_{local}^{k+1} array after step 1(d) (Fig. 2.7) instead of on \mathbf{b}_{local}^k array at step 1(a). That is, the global-concatenate operation is actually performed for the next iteration. Note that the first and/or last entries of the \mathbf{x}_{local} array may contain partial results at the end of step 1(b) due to the row sharing. Processors propagate these partial results to their \mathbf{b}_{local}^{k+1} arrays through steps 1(c) and 1(d). So, the first and/or last entries of the \mathbf{b}_{local}^{k+1} array may contain partial results just before the global-concatenate operation modified to handle these partial results. The exchange and local concatenate structure of the modified global-concatenate operation is exactly the same as that of the conventional one. However, just after the concurrent exchange step over channel j , processors whose j^{th} bit of their processor ids are 1(0) add the last(first) entry of the received array to the first(last) entry of their local array in addition to proper local concatenate operation if this location contains partial result. The concurrent addition operation after the exchange step over channel j corrects the partial result corresponding to the shared rows between successive processors of hamming-distance “ $j+1$ ” for $j=0,1,\dots,\log_2 P-1$. The proposed modification introduces an overhead of $(t_{tr}+t_{add})\log_2 P$ to each global-concatenate operation. Since $t_{su} \gg t_{tr}$ in medium-to-coarse grain parallel architectures (e.g., iPSC/2), the modified global-concatenate scheme performs much better than the single shift-and-add scheme. In **SCG** method, similar approach can be followed to incorporate the extra communication overhead due to the shared rows into global inner-product operation at step 2(a)-(b) in Fig. 2.9.

2.7 Experimental Results

The algorithms discussed in this chapter were implemented (in C language) on a 4-dimensional Intel iPSC/2 hypercube multicomputer. These algorithms were tested on

Table 2.1: Relative performance results in parallel execution times (in seconds) of different parallel algorithms for the form-factor computation phase. N is the number of patches in the scene and M is the number of non-zero entries in the form-factor matrix.

scene		P	static assignment			demand driven
			patch circulation	storage sharing		
N	M		random	random	tiled	
2600	1804647	16	1560.0	1193.6	1539.4	1149.9
		8	3046.4	2380.7	3024.9	2299.5
2208	1468539	16	1227.8	977.5	1203.5	929.1
		8	2383.4	1911.2	2365.5	1857.3
1728	746779	16	757.6	565.5	751.0	530.2
		8	1450.9	1099.0	1482.3	1059.0
		4	2719.5	2161.5	2909.9	2110.8
1412	461947	16	564.9	443.6	535.4	423.9
		8	1078.1	867.1	994.0	843.9
		4	2032.6	1700.7	1923.0	1684.9
		2	3764.7	3399.2	3594.3	3365.3
1000	342003	16	322.8	263.7	309.9	251.2
		8	616.1	512.3	621.8	499.7
		4	1173.8	1012.5	1149.3	996.8
		2	2191.2	2014.4	2223.8	1993.0
886	303146	16	274.4	224.8	254.8	215.5
		8	519.1	439.0	492.3	428.9
		4	997.2	870.2	955.7	853.9
		2	1858.5	1719.0	1858.3	1708.1

different room scenes containing various objects discretized into different number of patches ranging from 496 to 2600 patches. In the tables, N is the number of patches in the scene, M is the number of non-zero entries in the form-factor matrix, and P is the number of processors.

Table 2.1 illustrates the relative performance results of different parallel algorithms for the form-factor computation phase. The execution times of different algorithms are also illustrated in Fig. 2.10(a) for 16 processors. Parallel timing results for the random assignment scheme denote the average of 5 different executions for different random assignments. As is seen in the table, storage sharing scheme gives better performance results compared to the patch circulation scheme. In the storage sharing scheme, random

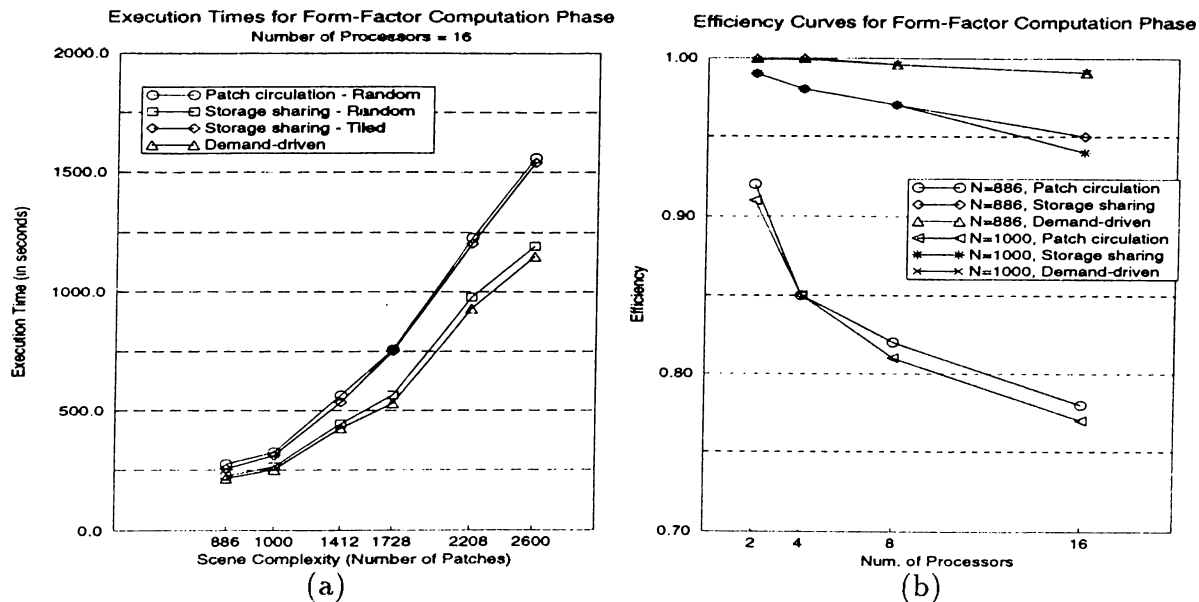


Figure 2.10: Form-factor computation phase. (a) Execution times for different schemes on 16 processors. (b) Efficiency curves for different schemes.

decomposition yields better load balance than the tiled decomposition as is expected. However, tiled assignment in storage sharing scheme yields better results in most of the test instances (e.g., 15 out of 19) than the random assignment in patch circulation due to the decrease in communication overhead. As seen in Table 2.1 and in Fig. 2.10(a), demand driven scheme always performs better than the static assignment scheme due to better load balance. Note that experimental timing results for some of the instances are missing for small number of processors due to insufficient local memory sizes. The sequential timings could only be obtained for the smallest size scenes with $N = 886$ and $N = 1000$ as 3418.7 seconds and 3981.3 seconds, respectively. The efficiency curves for these scenes are illustrated in Fig. 2.10(b). Demand-driven scheme yields almost 0.99 efficiency even for these two small scenes on a hypercube with 16 processors.

The effects of the *assignment granularity* on the form-factor computation and solution phases for the demand driven assignment scheme are also experimented. The results of these experiments are displayed in Figure 2.11. The *assignment granularity* denotes the number of patches assigned and sent to the requesting idle processor by the host processor. Small assignment granularity (e.g., single patch assignment) gives better performance in parallel form-factor computation phase due to the better load balance in

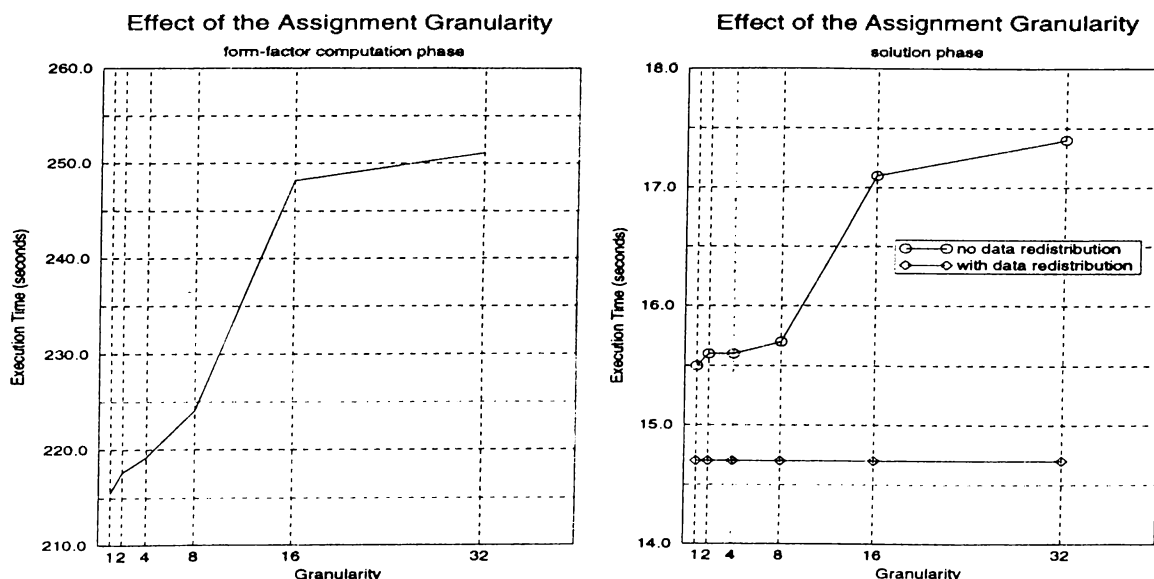


Figure 2.11: The effect of the *assignment granularity* on the performance (execution time in seconds) of the demand driven scheme for $N = 886$, $P = 16$.

spite of the increased communication overhead. Therefore, we can deduce that the calculation of a single form-factor row is computationally intensive and hence load balance is a more crucial factor than the communication overhead in this scheme. As is also seen in the figure, a similar behavior is observed in the solution phase when redistribution of non-zero entries is not done. Note that higher granularity means a processor will generate more rows for a single request. Hence, the number of non-zero entries in the local slices of \mathbf{F} matrix in each processor may be substantially different, incurring more load imbalance in the solution phase for higher granularity values. As is expected, when data redistribution is applied, the execution time of the solution phase remains constant irrespective of the assignment granularity.

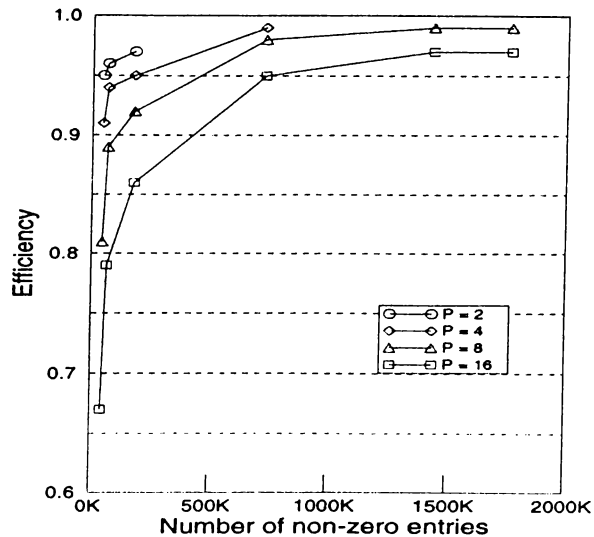
Table 2.2 illustrates the performance comparison of various schemes in the solution phase along with the associated overheads. As is expected, data redistribution achieves performance improvement due to better load balancing in spite of the preprocessing overheads. The overall performance gain will be much more notable for repeated solution operations as is required in lighting simulations since the data redistribution is to be performed only once for such applications. As is seen in this table, the time spent for renumbering and data redistribution operation is substantially smaller than even

Table 2.2: Parallel execution times (in seconds) of various schemes in the solution phase (using **GJ**) along with the associated overheads. TOT is the total execution time including overheads, i.e. $TOT = (\text{solution} + \text{preprocessing})$ time. N is the number of patches in the scene and M is the number of non-zero entries in the form-factor matrix.

scene		P	static rand.	demand driven								
				no redistribution				redistribution				
			TOT	preproc. time	solution time		TOT	preproc. time		solution time		TOT
N	M			renum.	iter.	total		renum.	redist.	iter.	total	
1412	462K	16	28.3	0.149	0.371	28.2	28.3	0.138	0.170	0.346	26.3	26.6
		8	54.2	0.257	0.699	53.1	53.3	0.250	0.117	0.680	51.7	52.1
		4	104.9	0.484	1.368	104.0	104.5	0.477	0.130	1.350	102.6	103.2
		2	207.8	0.935	2.697	205.0	205.9	0.932	0.049	2.691	204.5	205.5
1000	342K	16	18.2	0.114	0.280	17.9	18.0	0.104	0.093	0.263	16.8	17.0
		8	34.8	0.195	0.523	33.5	33.7	0.189	0.117	0.514	32.9	33.2
		4	66.8	0.364	1.030	65.9	66.3	0.359	0.085	1.020	65.3	65.7
		2	131.3	0.708	2.045	130.9	131.6	0.702	0.024	2.042	130.1	130.9
886	303K	16	16.0	0.099	0.242	15.5	15.6	0.095	0.074	0.230	14.7	14.9
		8	30.2	0.173	0.463	29.6	29.8	0.167	0.078	0.452	28.9	29.2
		4	58.2	0.320	0.903	57.8	58.2	0.316	0.038	0.894	57.2	57.6
		2	115.3	0.627	1.811	115.9	116.5	0.617	0.030	1.783	114.1	114.8

the solution time per iteration and yields considerable performance increase during the parallel solution. For example, by spending almost 0.6% of solution time in data redistribution, we reduce the total solution time by almost 7.1% on 16 processors for the scene with $N = 1412$ patches. The relative performance gain achieved by adopting data redistribution is expected to increase with increasing number of processors. Table 2.2 also illustrates the decrease in the execution time of the parallel renumbering operation whenever the data redistribution operation is performed. This is due to the fact that load balance metric in both parallel renumbering and matrix-vector product operations are exactly the same, i.e. equal number of non-zero matrix elements in each processor.

Table 2.3 illustrates the performance comparison of the *Gauss-Jacobi* and *Scaled Conjugate-Gradient* methods for the parallel solution phase. Note that experimental timing results for some of the instances on small number of processors are missing due to the insufficient local memory size. However, sequential timings for the scenes with

Figure 2.12: Efficiency curves for the **SCG** method.

$N = 1728, 2208, 2600$ patches are estimated using the sequential complexity expressions given in Eq. (2.7) and (2.11) and using $t_{calc} = 5.87$ microseconds for the sake of efficiency computations. The number of iterations denote the total number of iterations required for convergence to the same tolerance value (5×10^{-6}) for three color bands (i.e., red, green, blue). As is seen in Table 2.3, an individual **SCG** iteration takes more time than that of **GJ** iteration. However, the **SCG** method converges much faster than the **GJ** method as is expected. Therefore, we recommend the parallel **SCG** method for the solution phase. Figure 2.12 illustrates the efficiency curves of the **SCG** method. As is seen in this figure, the efficiency remains above 86% for sufficient granularity (i.e., $M/P > 11148$).

2.8 Conclusions

In this work, a parallel implementation of gathering method on hypercube-connected multicomputers has been discussed. Several algorithms have been developed for the form-factor computation and solution phases.

In the form-factor computation phase, it has been illustrated that it is possible to reduce the interprocessor communication by sharing the memory space for rows of the

form-factor matrix with global patch data. It is also observed that demand-driven approach, in spite of its extra communication overhead, achieves better load balancing and hence better processor utilization. Therefore, we conclude that demand-driven approach is more suitable for the form-factor computation phase.

In the solution phase, almost perfect load balance has been achieved by an efficient data redistribution scheme. This scheme brings negligible communication overhead while maintaining much better load balancing during the iterations. The powerful Scaled Conjugate-Gradient method has been successfully applied in the solution phase. We conclude that the Scaled Conjugate-Gradient method is a much better alternative to the conventional Gauss-Jacobi method for the parallel solution phase.

Although the target architecture is hypercube-connected multicomputer in this work, algorithms developed can be adopted for other interconnection topologies (e.g., rings and meshes). Static assignment schemes for the form-factor computation phase need no modification if a ring can be embedded on the target architecture. Demand-driven scheme uses the host processor of iPSC/2 to process patch requests from node processors. If there is no separate host processor in the parallel machine, one of the node processors can process patch requests from other processors in addition to calculating form-factor values. It is likely that this additional work on that node processor will degrade the performance. However, it can be expected that performance decrease will not be high because processing a request is not very computation intensive. It only involves selecting a patch from global patches and sending it to the requesting node processor. In the solution phase, communication structures of the global operations in parallel **GJ** and **SCG** schemes need to be modified for the target architecture.

Table 2.3: Performance comparison of parallel *Gauss-Jacobi* and *Scaled Conjugate-Gradient* methods (1* denotes the estimated sequential timings). Timings are in seconds. N is the number of patches in the scene and M is the number of non-zero entries in the form-factor matrix.

scene		P	Gauss-Jacobi			Scaled Conjugate-Gradient		
N	M		exec. time		# of iter.	exec. time		# of iter.
			total	iter.		total	iter.	
2600	1804647	16	124.0	1.35	92	54.1	1.39	39
		8	246.2	2.68	92	106.7	2.74	39
		1*	1957.6	21.28	92	837	21.47	39
2208	1468539	16	102.1	1.10	93	46.4	1.13	41
		8	202.6	2.18	93	91.3	2.23	41
		1*	1610.6	17.32	93	716.4	17.47	41
1728	746779	16	51.5	0.57	91	23.6	0.59	40
		8	101.6	1.12	91	46.1	1.15	40
		4	202.0	2.22	91	91.0	2.28	40
		1*	803.4	8.83	91	358	8.95	40
1188	178374	16	12.6	0.15	87	6.1	0.16	38
		8	24.2	0.28	87	11.4	0.30	38
		4	47.4	0.55	87	21.9	0.58	38
		2	93.9	1.08	87	43.1	1.13	38
		1	186.7	2.15	87	83.5	2.20	38
880	45889	16	4.1	0.05	89	2.4	0.06	41
		8	7.1	0.08	89	4.0	0.10	41
		4	13.3	0.15	89	7.1	0.17	41
		2	26.1	0.29	89	13.6	0.33	41
		1	51.4	0.58	89	25.8	0.63	41
496	66900	16	4.8	0.06	83	2.5	0.07	38
		8	8.8	0.11	83	4.4	0.12	38
		4	17.2	0.21	83	8.4	0.22	38
		2	33.8	0.41	83	16.4	0.43	38
		1	67.0	0.81	83	31.5	0.83	38

Chapter 3

Polygon Rendering: Overview and Related Work

In this chapter, an overview of sequential polygon rendering is given and previous works on parallel polygon rendering algorithms are summarized.

3.1 Sequential Polygon Rendering

In simple terms, polygon rendering is the process of displaying three dimensional objects and scenes composed of polygons. It is basically a pipeline of operations applied to the polygons and objects in the scene to produce a realistic picture on the computer screen. This pipeline is illustrated in figure 3.1.

3.1.1 Reading Environment Description

At this step, the environment description is read into the computer. The description of the environment is converted into a suitable form to perform other operations in the pipeline. For example, if there are objects or surfaces in the environment that are not planar polygons, these objects and surfaces are approximated by polygons. A polygon is defined by a set of vertices, a set of vertex normals, a surface normal and reflectivity values for red, green and blue colors. Figure 3.2 illustrates a polygon with 5 vertices. The set of vertices is represented as an ordered list of points, with 3-dimensional spatial coordinates as (x, y, z) in *world coordinate system*. Two successive vertices in

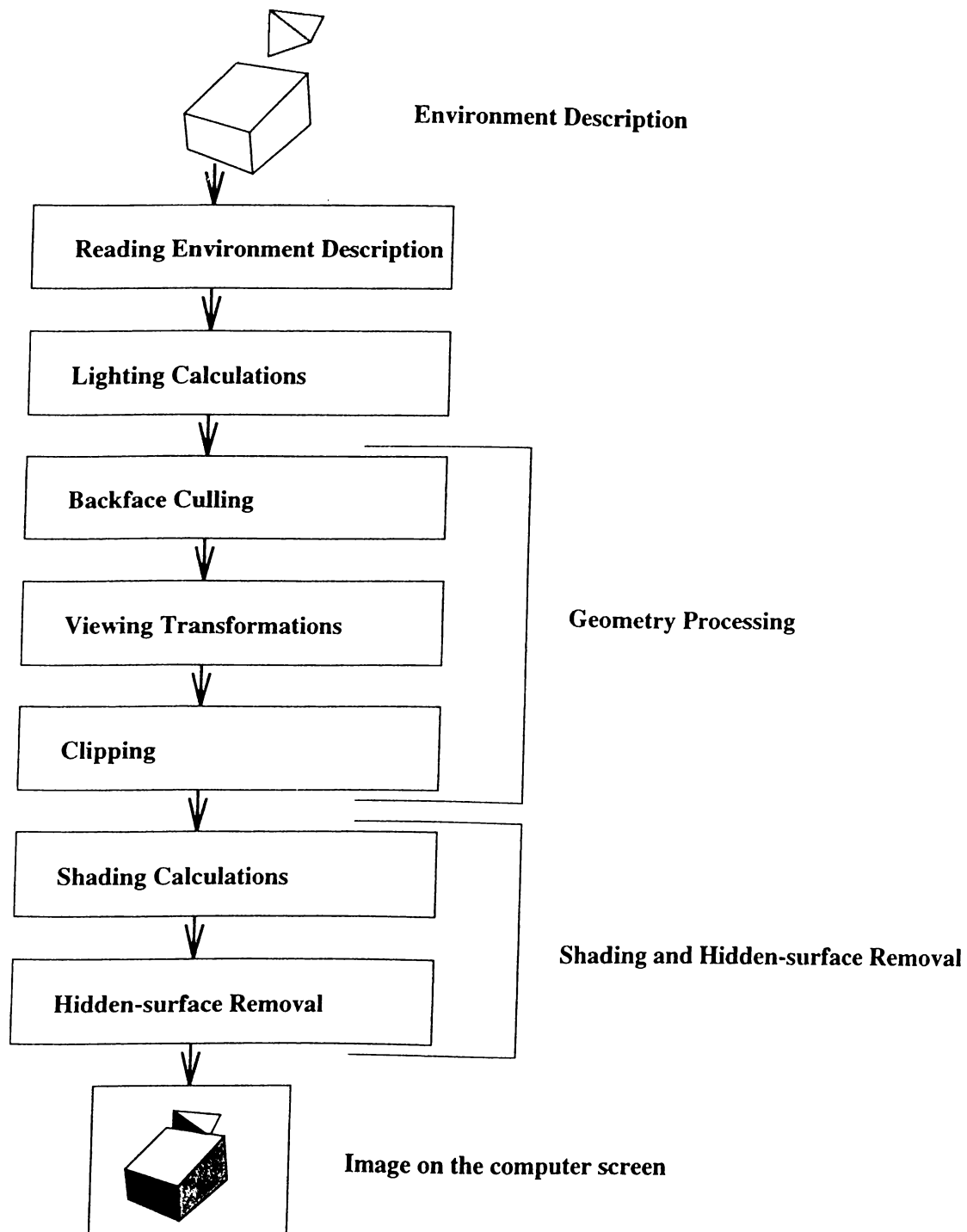


Figure 3.1: The polygon rendering pipeline.

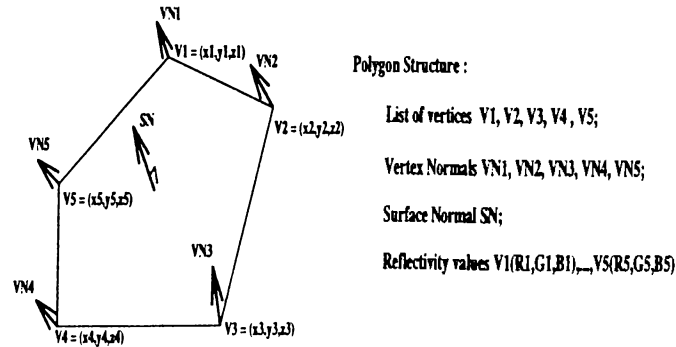


Figure 3.2: A polygon with 5 vertices.

the list define an edge of the polygon. Although the surface normal of the polygon is constant over the surface of the polygon, vertex normals need not be aligned with the surface normal. If the polygon is a part of a surface or object, a vertex normal actually indicates the surface normal of the object or surface at that vertex. Similarly, vertex reflectivity values indicate the reflectivity values of the object or surface at the corresponding vertices.

3.1.2 Lighting Calculations

The light-object interactions are calculated at this step to simulate the propagation of light in the environment. Simple methods or more realistic methods can be used. If simple methods, which do not account for the reflected light, are utilized, this step can be done after backface culling. Backface culling eliminates polygons that are facing away from the viewing direction. Therefore, lighting calculations are avoided for those polygons. Vertex normals and reflectivity values at the vertices of the polygon are used in lighting calculations in simple methods. If more realistic methods are used, like radiosity that accounts for the reflected light as well, this step should be performed before backface culling.

3.1.3 Geometry Processing

Geometry processing is applied to transform polygons from world coordinate system to viewing coordinate system (Fig. 3.3) and to eliminate polygons that are not visible

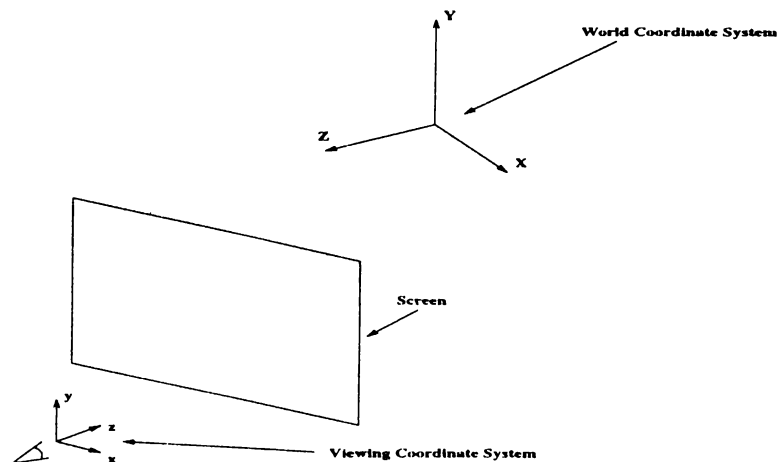


Figure 3.3: World and viewing coordinate systems.

from the viewing direction and orientation. First, polygons that are facing away from the viewing direction is eliminated. Backface culling is performed using the surface normal of the polygon and the surface normal of the screen. Remaining polygons are transformed from world coordinate system to viewing coordinate system. The viewing coordinate system describes the viewing direction, position and orientation of the screen. Viewing transformations are performed by creating a 4×4 *viewing matrix* (or transformation matrix) from viewing parameters (e.g., orientation of the screen, distance of the screen from the origin of the world coordinate system, etc.) and multiplying vertices of each polygon by this matrix. Note that each polygon vertex is represented in cartesian coordinate representation, i.e., coordinates of the vertex is represented as (x, y, z) . In order to multiply coordinates of a vertex with viewing matrix, *homogeneous coordinate* representation is adopted, i.e., coordinates of a vertex is represented as (x', y', z', h) such that $x = \frac{x'}{h}$, $y = \frac{y'}{h}$ and $z = \frac{z'}{h}$. Perspective transformation is applied to give realism to images. Polygons are clipped to screen boundaries [85] to eliminate parts of the polygons that are not visible from the viewing direction and orientation.

3.1.4 Shading and Hidden-surface Removal

Hidden-surface removal and smooth shading of the transformed polygons are performed to produce a realistic image of the environment. Shading and hidden-surface removal are

the most time consuming steps of the polygon rendering pipeline. In practice, shading calculations and hidden-surface removal operations are done concurrently in a single step rather than separate steps as is illustrated in Fig. 3.1.

If simple methods are used for lighting calculations, smooth shading of the polygons can be performed by using either Gouraud shading method or Phong shading method. In Gouraud shading method [34], the light-polygon interactions are calculated at the vertices, producing intensity values. These intensity values are interpolated over the polygon surface to shade the polygon. In Phong shading method [68], vertex normals are interpolated over the polygon surface and intensity values at a point on the polygon are calculated using the interpolated normal at that point. If more realistic methods are used, smooth shading of polygons can be done in a way similar to Gouraud shading method.

Hidden-surface removal determines which polygons are visible at certain screen locations (pixel locations). The hidden-surface removal process is a kind of sorting operation [86] to determine the visibility and visible parts of the polygons. Basically, polygons are sorted by their distance (z) to the screen. The overhead of sorting is reduced by utilizing some coherency property existing in the environment such as image-space coherency. The algorithms in hidden surface removal can be classified into two groups as *object-space* algorithms and *image-space* algorithms [86, 95, 77]. In object-space algorithms, visibility of polygons is determined in 3-dimensional space. Since these algorithms operate in continuous domain, visibility calculations can be performed at any precision. Objects in the environment are compared with each other to determine the visible parts. After the visible parts of the polygons are determined, these parts are displayed on the screen. In image-space algorithms, visibility is determined on the screen, on which the image of the environment is generated. The sorting operation is done at pixel locations. In order to accomplish this, each polygon is projected onto the screen. Distance and color values are generated for screen coordinates that are covered by the projected polygon. This process is called *rasterization* or *scan-conversion*. Note that screen coordinates are discrete quantities. Therefore, unlike object-space algorithms, image-space algorithms operate in a discrete domain. The distance values of the pixels generated for the same pixel location are compared and pixel closest to the screen (foremost pixel) is stored into

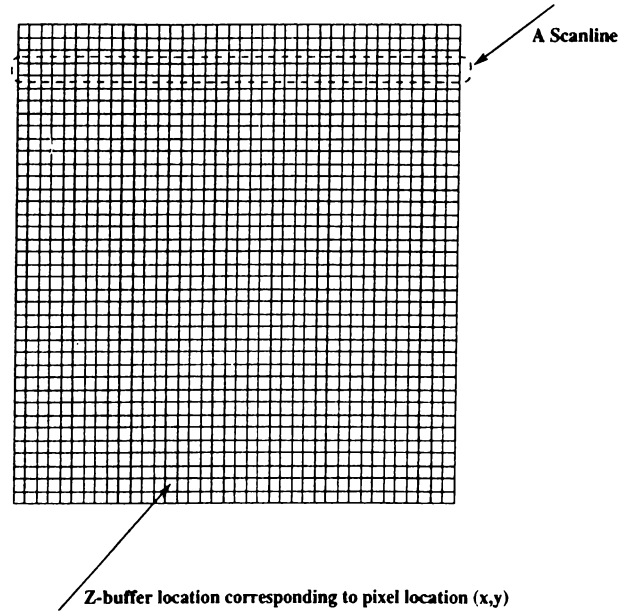


Figure 3.4: The z-buffer array.

the frame buffer (displayed on the screen). Image-space algorithms are more popular than object-space algorithms due to better utilization of coherency and wider range of applications. Use of coherency in image-space allows incremental calculation of distance and color values over the surface of the polygon.

Two images-space algorithms are quite popular in computer graphics rendering. These algorithms are called *z-buffer* and *scanline z-buffer* algorithms. Main difference between these algorithms is the order in which polygons are rasterized. The *z-buffer* algorithm performs rasterization operations in polygon-order, whereas *scanline z-buffer* algorithm performs rasterization in scanline-order.

Z-buffer Algorithm

In *z-buffer* algorithm, a 2-dimensional array, called *z-buffer*, is used to perform hidden surface removal (Fig. 3.4). There exists one-to-one correspondence between *z-buffer* and the screen. The entry at array location (x, y) corresponds to the pixel location (x, y) on the screen. Distance value of the foremost polygon at this pixel location is stored into the *z-buffer*. Initially, distance values at all *z-buffer* locations are set to infinity.

Each projected polygon is rasterized one-by-one. The color and distance values are interpolated over the surface of the polygon to generate pixels with distance and color information. The distance value of the pixel generated for pixel location (x, y) on the screen is compared with z-buffer position (x, y) . If the distance of the pixel generated is smaller than the value at z-buffer location (x, y) , the distance value of the pixel is stored into that location and the color of the pixel location (x, y) on the screen is set to the color of pixel generated.

Scanline Z-buffer Algorithm

In scanline z-buffer algorithm, hidden-surface removal is performed in scanline-order. The algorithm proceeds from one scanline to the next starting from the lowest numbered scanline on the screen. A *scanline* is a row of pixels on the screen (similarly on z-buffer array as is illustrated in Fig. 3.4). Polygons whose projections intersects the current scanline are processed. A one dimensional array, called *scanline z-buffer*, is used to perform hidden surface removal. There is one-to-one correspondence between array locations and pixel locations on the current scanline. The array location x stores the distance of foremost polygon at pixel location x on the current scanline. The initial step of the algorithm inserts polygons into *y-bucket structure*, which is an array (of size N , where N is the y-dimension of the screen) of linked lists. There exists one-to-one correspondence between array locations and scanlines on the screen. A polygon is inserted to the linked list at y-bucket location which corresponds to the lowest scanline that intersects the projection of polygon. After this initialization step, hidden-surface removal is performed in scanline-order starting from the lowest numbered scanline. Polygons in the y-bucket of the current scanline and polygons whose y-extend covers this scanline are processed. Edge intersection of these polygons with the current scanline is found. These intersections create line segments, called *spans*, which are rasterized one-by-one generating pixels for pixel locations on the current scanline. As in z-buffer algorithm, the distance value of the pixel generated is compared to the distance value at corresponding scanline z-buffer array. If the distance of the pixel is smaller, its distance value is stored into array location x in scanline z-buffer and the color of the pixel location on the screen

is updated. Scanline z-buffer algorithm utilizes image-space coherency from scanline-to-scanline and pixel-to-pixel. Edge intersections are found by incremental calculations from scanline-to-scanline. Distance and color calculations are also done incrementally within a span from pixel-to-pixel in the current scanline. Scanline z-buffer is re-initialized by setting distance values in all array locations to infinity before processing the current scanline.

3.2 Previous Works on Parallel Polygon Rendering

In this section, previous works on parallel polygon rendering are summarized. First, we introduce a taxonomy of the parallel polygon rendering algorithms. After the presentation of the taxonomy, previous works, which are classified with respect to taxonomy, on parallel polygon rendering are described.

3.2.1 A Taxonomy of Parallelism in Polygon Rendering on Distributed-Memory Multicomputers

There are various classifications for parallelism in polygon rendering [21, 99, 61, 18]. In this thesis, a taxonomy, based on the domain that is partitioned among the processors, is given. The screen, on which the result of the rendering is displayed, constitutes the output domain (image-space domain) of the rendering process. The input domain, which is also referred to as object-space domain, of the rendering process is the input data set defined in 3-dimensional space.

Image-space Parallelism

The domain of decomposition in this parallelism approach is the output domain of the rendering process. The screen is divided into subregions and each processor is assigned one or more of the subregions to render (Fig. 3.5). The object database is also partitioned according to screen subdivision and primitives are re-distributed to respective processors. The term *re-distribution* is used to indicate that primitives may already be in local memories of the processors. Each processor may require the knowledge of

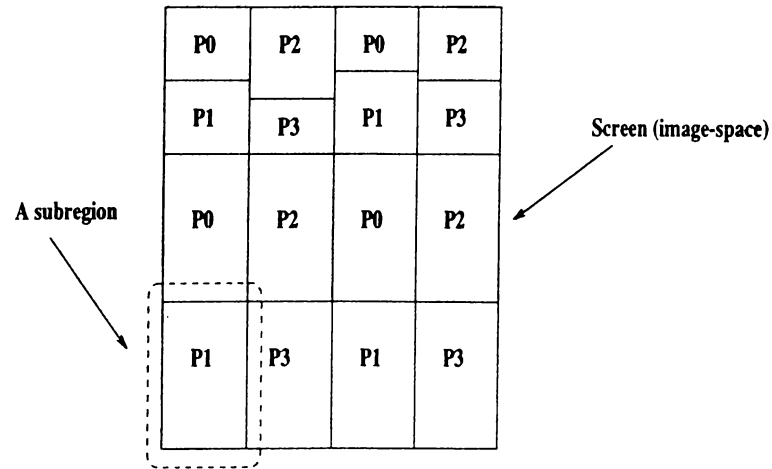


Figure 3.5: An example of image-space parallelism. The screen is partitioned and subregions are assigned to processors (**P0**, **P1**, **P2**, **P3**).

how screen is subdivided and assigned to other processors. Primitives that belong to more than one subregion are duplicated or divided into smaller primitives. Therefore, image-space algorithms may cause an increase in the number of primitives in the overall system. After the primitive distribution, each processor performs rendering of subregions assigned without further inter-processor communication. Primitives are re-distributed among processors when viewing point and orientation changes and/or screen is subdivided again and/or subregions are re-assigned to processors. In a recent paper [61], image-space parallelism has also been subdivided into two subclasses as *sort-first* and *sort-middle* according to when in the *polygon rendering pipeline* the primitives are re-distributed. In *sort-first* approach, a simple processing is performed on primitives before *geometry processing* of rendering pipeline to find the regions each primitive belongs to. Then, primitives are distributed to respective processors. Receiving processors perform geometry processing, shading and hidden-surface removal. In *sort-middle* approach, primitives are re-distributed after sending processor performs geometry processing on the primitives. All local primitives are transformed and clipped to subregion boundaries in each processor. Transformed and clipped primitives belonging to regions assigned to other processors are transmitted to corresponding processors. Receiving processors only perform shading and hidden-surface removal operations.

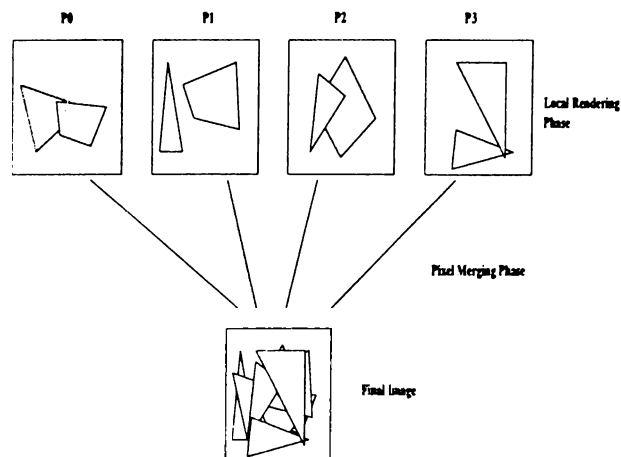


Figure 3.6: An example of object-space parallelism.

Object-space Parallelism

In object-space parallelism, the domain of decomposition is the input domain of the rendering process. The primitives (polygons, objects, etc.) that constitute the environment are divided into groups and distributed among the processors. Usually, primitives are *not* re-distributed when viewing position and orientation, or screen resolution changes. Primitive re-distribution may be necessary if larger primitives are divided into smaller ones or smaller primitives are combined into larger primitives. Division and combination operations are application dependent and are not related to the parallel algorithm. Note that each primitive is assigned to a unique processor in object-space parallelism. Therefore, unlike image-space parallelism, the original number of primitives remains constant. After distribution of primitives, each processor performs rendering of local primitives, thus producing partial images. After this *local rendering* phase, partial images in each processor are merged to obtain the final picture because primitives in different processors may contribute to the same pixel location on the screen. This *pixel merging* (or *image composition*) phase is performed by exchanging local image buffers fully or partially over the inter-connection network. Figure 3.6 illustrates an example of object-space parallelism on four processors. Object-space parallelism is also called *sort-last* approach [61].

Comparison of Image-space Parallelism and Object-space Parallelism

Both approaches have been explored and applied for parallel computer graphics rendering by researchers. This section discusses advantages and disadvantages of image-space and object-space parallelism with respect to utilization of distributed-memory multicomputers.

The main source of inter-processor communication in image-space parallelism is the re-distribution of primitives among the processors according to screen subdivision and assignment of screen regions to processors. Note that re-distribution of primitives also takes place when viewing position and orientation changes even if the shape and number of screen regions and assignment of regions to processors do not change. The volume of communication depends on the number of primitives re-distributed and the amount of information to represent a primitive. It is irrespective of the resolution of the screen and the amount of information stored at each pixel. Advantage of the image-space parallelism is that no inter-processor communication occurs when screen regions are assigned to processors statically and viewing parameters do not change. This type of application occurs in lighting simulations in radiosity and in volume rendering when scientist wants to visualize the volume using different transfer functions, which map the contribution of data values to color values. The image-space parallelism can also be advantageous for interactive applications where the viewing parameters change gradually and only few primitives are likely to be re-distributed. However, for large number of primitives, even a small perturbation of the viewing parameters may result in re-distribution of many primitives.

The main source of inter-processor communication in object-space parallelism is the pixel merging phase. In pixel merging phase, the pixels on the screen are exchanged among processors. Therefore, the volume of communication in object-space parallelism depends on the resolution of the screen, number of pixels actively covered by polygons in each processor, and amount of information stored in each pixel location. It is irrespective of the amount of information used to represent primitives. However, it depends on the number of primitives and the size of primitives in an indirect way. When the number or size of the primitives increase, it is likely that the number of pixels covered by these primitives will increase, resulting in more pixels to be exchanged in the pixel merging

phase. Advantage of the object-space parallelism is that primitives are not re-distributed even viewing parameters change.

In image-space parallelism, load balancing heuristics operate on the screen to achieve an even distribution of load. Since the image-space is a discrete environment, the accuracy of load balancing depends on the resolution of the screen. Achieving the best assignment of regions to the processors is another problem that exists in image-space parallel algorithms. In object-space parallelism, load balancing heuristics operate on object-space to achieve a good load balance in local rendering phase. However, load balancing in pixel merging phase is also another important issue in object-space parallelism. Note that pixel merging is performed in 2-dimensional image-space. If algorithms, which only exchange pixels covered by local polygons, are utilized in pixel merging phase, the distribution of pixels on the screen affects the load distribution in this phase. As a result, load balancing heuristics that operate on image-space subdivisions (as in image-space parallelism) are also needed to achieve load balancing in pixel merging phase.

In image-space parallelism, a primitive is duplicated in different processors if the primitive overlaps regions assigned to different processors. As a result, the number of primitives in the overall system increases in image-space parallelism. Since primitives are assigned to unique processors in object-space parallelism, primitive duplication does not occur.

The type of algorithms used for rendering of the primitives and the characteristics of application may impose serious problems on object-space parallelism due to pixel merging phase. For example, anti-aliasing techniques may introduce more complicated implementation of pixel merging phase. In addition, the view sorted composition restriction in volume rendering may impose complications in both subdivision and assignment of primitives to processors and on the algorithms used in pixel merging phase. These complications usually do not occur in image-space parallelism since after the assignment of regions and primitives to processors, each processor effectively runs a sequential rendering algorithm.

3.2.2 Previous Works on Parallel Polygon Rendering

This section describes the previous works, which are classified with respect to the taxonomy described in the previous sections, on parallel polygon rendering. In image-space parallelism, subdivision of the screen and load balancing are the key issues that were covered by the researchers. Hence, summaries of previous work in image-space parallelism focus on how screen is divided and how load balancing is performed in these works. In object-space parallelism, the focus in the summaries of these works will be on how pixel merging phase is implemented by different researchers.

Previous Works on Image-Space Parallelism

Mueller [65] presents a sort-first [61] parallel rendering algorithm for interactive applications. Static and adaptive division of the screen is examined for load balancing. In static subdivision scheme, the screen is subdivided into rectangular regions which are assigned to processors in round-robin fashion using a scattered assignment for load balancing. In this assignment strategy, adjacent regions are assigned to different processors such that processor i is assigned regions i , $i + P$, $i + 2P$, and so on. Here, P denotes the number of processor in the multicomputer. In adaptive subdivision scheme, the screen is subdivided adaptively using the distribution of triangles on the screen until the number of regions is equal to the number of processors. In order to find the distribution of triangles on the screen, a “fine mesh” is superimposed on the screen. The number of primitives, which cover the mesh cell, is counted for each mesh cell. An amount inversely proportional to the number of cells a primitive covers is added to corresponding mesh cell count to avoid errors caused by counting large primitives multiple times. A single processor collects counts from each processor and forms a summed-area table [20], which has the same resolution as the fine mesh. This processor divides the screen recursively in alternate directions at each step using the summed-area table. The summed-area table allows binary search to determine the division line. The screen subdivision information is broadcast to each processor so that primitives are re-distributed according to new subdivision. Adaptive subdivision exploits frame-to-frame coherence existing in interactive applications. Current frames distribution is used to perform subdivision for the

next frame. Static and adaptive subdivision schemes are evaluated experimentally using a simulator with respect to various factors such as number of regions, mesh resolution, effect of the number of processors.

Crockett and Orloff [19] present a parallel polygon rendering algorithm on Intel iPSC/860. Their algorithm distributes triangles to processors in round-robin fashion using a scattered assignment scheme. Each processor receives an even number of triangles. The screen is subdivided into horizontal bands of equal number of consecutive scanlines and each region is assigned to one processor. Local triangles in each processor are transformed, clipped and light interactions are calculated. Resulting 2-dimensional triangles are split into trapezoids at the boundaries of horizontal bands. Each trapezoid is inserted into a message buffer to send it to the processor that owns the horizontal band, to which the trapezoid belongs. The receiving processor performs rasterization of trapezoids and hidden-surface removal. In order to have better utilization of processors, trapezoid generation and transmission is multiplexed with rasterization of received trapezoids. In the paper, the impact of the length of the message buffer and communication overheads are examined analytically and experimentally. Load balancing issues are not discussed.

Ellsworth [26] proposes a parallel rendering algorithm for interactive applications on an Intel Touchstone Delta multicomputer. The screen is divided into equal size rectangular regions (close to square to decrease the number of polygons shared between regions) which are distributed to processors using a greedy multiple-bin-packing heuristic. Work load in each region is taken to be the number of polygons in that region. The regions are sorted in decreasing polygon counts. Starting from the region with highest polygon count and continuing in sort order, each region is assigned to the processor which currently has the minimum work load. After assignment of regions to processors, each processor performs geometry processing on its local polygons. During geometry processing each polygon is classified according to the regions it overlaps. After geometry processing, each polygon is sent to the processor, which owns the region(s) the polygon overlaps, for rasterization and hidden-surface removal. The algorithm utilizes frame-to-frame coherence to achieve an even load distribution among processors. Polygon distribution in the current frame is used to find assignment of regions in the next frame. During

rasterization step, each processor finds local polygon counts at each region using the local polygons. The polygon counts are directed to a single processor (processor 0 in the paper) in a summing tree to find global polygon counts in each region. Processor 0 performs region assignments and assignment information is broadcast to all processors in the multicomputer.

Whitman [99, 100] presents several algorithms for a BBN Butterfly shared-memory multiprocessor. This architecture provides a distributed shared memory in the form of memory boards associated with each processor. Processors access to the shared memory locations through a network called Butterfly switch. The software library provides a task generation mechanism that generates the next task to be assigned to processors dynamically [99, 11]. In his work, data non-adaptive, data adaptive, and task adaptive schemes are proposed and evaluated with respect to various factors such as communication overhead and load balancing. In the data adaptive and non-adaptive schemes, processors request task from task generator when they become idle. In the first non-adaptive scheme, each scanline on the screen is designated as a task. In the second scheme, the screen is divided into equal size rectangular regions, each of which is considered as a task. Various strategies are also presented to access the data associated with each task. In data adaptive scheme, screen is subdivided adaptively using polygon distribution on the screen. A 2-dimensional mesh is superimposed on the screen space and polygon counts are calculated in each mesh cell by using bounding boxes of the polygons. Adjacent mesh locations are combined hierarchically in a tree. Each node of the tree stores the number of polygons in the corresponding combined region. This tree is traversed in top-down fashion by splitting the region with the largest number of polygons until a desired number of regions is reached. In his work, the number of regions is taken to be ten times the number of processors. The top-down traversal of the tree is done sequentially on a single processor. After regions are created, each region is assigned to processors dynamically as in non-adaptive schemes. In addition to data partitioning algorithms, a task adaptive scheme is presented. In this scheme, idle processors share the work load of heavily loaded processors.

Roble [76] presents a scanline z-buffer algorithm for iPSC hypercube. A separate processor, called the cube manager (host) of the hypercube, reads polygon data and

performs geometry processing on the polygons. Afterwards, polygons are distributed to node processors in round-robin fashion using scattered assignment scheme. Initially, screen is subdivided into equal size rectangular regions with each region assigned to a processor. Processors obtain the polygon counts in each region and these counts are transmitted to cube manager. The cube manager combines lightly loaded contiguous regions into one region and divides heavily loaded regions into two subregions. After subdivision and combination operations, each processor is again assigned a single region. Polygon counts in each region only indicates the work load of the region but no information is provided on the distribution of polygons. Hence, no information is provided on how to find the optimal division line in a region. As is mentioned in the paper, the load balancing step may be repeated multiple times to obtain a better distribution. After cube manager performs subdivision of the screen, screen subdivision information and region assignments are broadcast to all processors and polygons are re-distributed according to new subregions.

Highfield and Bez [40] present and empirically compare parallel implementations of four rendering algorithms; recursive subdivision, scanline z-buffer, painter's, and z-buffer algorithms [77]. Their target architecture is a distributed-memory multicomputer composed of transputers with one "master" processor and a "chain" of "worker" processors. In parallel recursive subdivision algorithm, each worker processor is assigned a subregion of the screen to execute sequential recursive subdivision rendering algorithm. Master processor transmits polygon data to workers through the chain so that each worker receives a local copy of all polygons. Scanline z-buffer algorithm is parallelized by scattered assignment of the scanlines to processors in round-robin fashion. Polygon data is passed down the chain to worker processors as in recursive subdivision algorithm. In parallel z-buffer algorithm, two implementations are considered. Initial implementation is an object-space parallel approach as it partitions the polygon data among processors for rasterization. Each processor scan-converts the local polygons and sends rasterized polygon information to a single processor (screen processor) to do the hidden-surface removal. The single processor to perform the hidden-surface removal becomes a bottleneck that degrades the performance of the algorithm. An alternative implementation, which exploits image-space parallelism is devised. As in scanline z-buffer algorithm, scanlines

are scattered and each processor performs sequential z-buffer for scanlines assigned to it. The polygon data is passed down the chain as in scanline z-buffer algorithm. In the parallel painter's algorithm, again two implementations are considered as in the parallel z-buffer algorithm. First implementation partitions the polygon data among processors and each processor sorts the local polygons. The locally sorted lists are merged into a global sorted list by the master processor which then passes the global sorted list down the chain to workers. The worker processors scan convert the polygons in order, sending scanlines up the chain to screen processor for display. As in z-buffer algorithm, screen processor causes a bottleneck and degradation in the performance. An alternative algorithm which divides the screen among processors (as in parallel z-buffer algorithm) is also implemented.

Gupta and Fisher [36] present a parallelization of the scanline z-buffer on a linear array of processors. The linear array (which is rather a ring of processors) is divided into sets of equal number of processors. During the scanline processing, the current scanline is partitioned into equal number of pixels among the processor sets with each portion assigned to a different set. The polygon data is duplicated such that each set owns all of the polygons. Polygons are partitioned among the processors in a set so that each processor holds equal number of polygons. Similarly, scanline partitions are further divided equally among the processors in a set. After rendering of local polygons for the local scanline portion, each processor transmits the scanline partitions to the left neighbor in the ring. Receiving processor processes the local set of polygons for the received scanline portion. If there are k processors in a set, then these left-shifts of scanline portions are performed k times. Their algorithm can be considered to combine object-space and image-space parallelism. It is an image-space parallel algorithm because scanlines, thus screen, is divided among the processors. On the other hand, pixel information is also circulated between processors as in pixel merging phase of object-space algorithms. Their algorithm can also be considered as a "fine-grain" parallelization of scanline because portions of a single scanline is circulated in the linear array.

Li and Miguet [56] propose a parallel z-buffer algorithm implemented on a transputer architecture with reconfigurable interconnection network. The network is configured as a ring of transputers in their implementation. Initially, polygons are distributed to

processors so that each processor receives equal number of polygons. The image-space is divided into horizontal bands of equal number of scanlines. In order to improve the load balance, the bounding box of the environment is found by combining bounding boxes of polygons. Only the scanlines within the boundaries of the bounding box are divided into horizontal bands. Each band is assigned to a single processor. Polygons are re-distributed according to band assignments to processors.

Kaplan and Greenberg [46] discuss algorithms for a distributed-memory architecture with a central processor and node processors connected by a time-shared bus. The operation of the architecture is simulated in software. They present algorithms for scanline z-buffer and Warnock's area subdivision algorithms [77]. All of the polygon data is duplicated in the local memories of each processor. The screen is divided into group of scanlines for scanline z-buffer and rectangular regions for Warnock's algorithm. A central processor schedules tasks to parallel processors as they become idle.

Previous Works on Object-Space Parallelism

Cox and Hanrahan [17] propose a pixel merging algorithm developed for architectures with network broadcast capability. Their algorithm distributes polygons to processors in a round-robin fashion using scattered assignment scheme. Each successive polygon in the polygon database is assigned to successive processors in the architecture so that processor i receives polygons i , $i + P$, and so on. Here, P denotes the number of processors. Each processor applies polygon rendering pipeline to local polygons for the full screen. After this local rendering phase, pixel information (distance and color values) at each "active" pixel location, defined as the pixel location covered by at least one local polygon, is broadcast over the network to perform pixel merging phase. Starting from processor 1 and continuing in increasing processor number, processor k broadcasts over the network the local pixel information in local active pixel locations to a global frame-buffer (screen) and to processors $k + 1$, $k + 2, \dots, P$ that "snoop" the network to catch pixel information broadcast. Each snooping processor compares the distance values of received pixels with local pixels and eliminates hidden local pixels from further consideration. In this way, the number of pixels broadcast by the next processor is expected to decrease. The authors present an analytical discussion of expected case network traffic of their algorithm and

compare the analytical analysis with trace-driven simulations. No speedup figures are provided in the paper.

Scopigno *et al.* [81] present a parallel hidden-surface removal (HSR) paradigm based on divide-and-conquer approach. The hidden surface removal problem is solved by subdividing the problem into equal size subproblems recursively until the size of the subproblem is sufficiently small. In that case, HSR is done on the subproblem by “leafHSR processes”. The results of the leafHSR processes are then merged to obtain the final result. Authors present simulation results for tree-based and shared-memory architectures. In tree-based architecture model, each processor is assigned either to a leafHSR process or to a merge process. In shared-memory model, a scheduling processor assigns processors to leafHSR and merge processes. Message passing overhead and memory contention issues are not included in their simulations.

Li and Miguet [56] present an algorithm for transputers interconnected by a reconfigurable network. Their implementation configures the network as a tree structure. Polygon data is distributed to processors so that each processor receives an equal number of polygons. Pixel merging phase is done using the tree structure. In order to increase processor utilization and reduce memory requirements, the screen is divided into horizontal bands and processing of these bands are pipelined. Once a processor finishes the work on a band, it merges the results from its children in the tree and sends the merged band to its parent. In this way, while a processor processes k^{th} band of the screen, its parent processes the $(k - 1)^{st}$ band and its children process $(k + 1)^{st}$ band. In their implementation, ternary tree, binary tree and unary tree (ring) interconnection topologies are investigated for pixel merging phase.

Molnar *et al.* [62] present a object-space parallel rendering algorithm and architecture. In their work, partial images are merged in a pipelined “image-composition” network. After rendering of local polygons, full z-buffer in each processor is injected into the network, and each “compositor” in the pipeline network merges the partial image it receives and local partial image and directs the resultant full z-buffer to other compositors in the lower levels of the pipeline. In their paper, they present a hardware design to perform rendering and image composition (merging) operations.

In a very recent work, Lee *et al.* [53] present several algorithms for pixel merging

phase suited for 2-dimensional mesh multicomputers. Their target machine is Intel's Delta computer with 512 processors. In their schemes, they perform pixel merging in two stages. For this, the 2-dimensional mesh (with $r \times c$ processors) is organized as c independent rings, each consisting of r processors, in the rows and r independent rings, each consisting of c processors, in the columns. In the first stage, the full screen partial images in each processor are divided into r horizontal regions. These regions are merged concurrently in the rings in the rows. At the end of first stage, each processor has intermediate partial subimage ($1/r^{th}$ of the full screen image). In the second phase, the subimages in each processor are further divided into c horizontal regions. These regions are merged concurrently the rings in the columns to produce the final image. In their first scheme, regions of full z-buffer is circulated in the rings. In the second scheme, the volume of communication is reduced by sending bounding boxes that cover only active pixels. In these two schemes, screen regions are circulated in the rings by merging and forwarding received partial images to neighbour processors until they reach the destination processor. In their direct pixel forwarding scheme, the partial images are sent directly to destination processor. This scheme is also carried out in two stages. In the first stage, as in previous schemes, screen is divided into r horizontal regions and each processor in the ring is assigned a region. In each processor, a send queue is associated with each region. Processors store the active pixels generate during local rendering in the corresponding queue according to screen region. Pixel's x and y coordinates, color values, and z value are stored in the send queue. No local z-buffering is performed in this stage. That is, all generated pixels are stored into send queues. These send queues are directly transmitted to destination processors in the ring in the row. Each processor, then, z-buffers the received pixels to reduce the volume of communication for the next stage. In the next stage, active pixels in each processor are merged in the rings in the columns as in the first stage. Their last pixel merging scheme multiplexes local rendering and pixel merging computations. The pixel merging is done using direct pixel forwarding. However, processors keep fixed length buffers and during local rendering they send a buffer to destination processor when it is full. Thus, each processor switches between local rendering and pixel merging calculations. Lee et al. also address the load balancing in pixel merging phase. The subregions assigned to processors consist of

interleaved scanlines rather than consecutive scanlines for better load balance in pixel merging phase.

Other Previous Works

The algorithms presented in the previous works above are designed for coarse-grain multicomputers. Other works also exist in literature that exploit different approaches and different architectures. Some of these works is summarized below.

Theoharis and Page [91] give a parallelization approach for SIMD 2-dimensional processor arrays. The image space is mapped to processor array as a 2-dimensional grid. Assuming there are $P \times P$ processors in the SIMD array, the screen is partitioned into regions of $P \times P$ pixels each. Each processor is assigned a pixel from each region. The rendering operations are formulated as linear functions. The processor array performs rendering operations for a single polygon by evaluating these linear functions in parallel for the pixels in a region. Evaluation of linear functions in different regions are done by incremental calculations to utilize coherency.

Pineda [70] explores a similar approach to that of Theoharis and Page to rasterize a polygon by evaluating linear functions. The parallel algorithm presented utilizes a group of “interpolators”, each being responsible for evaluating linear functions for a single pixel within a contiguous block of pixels. No implementation of the algorithm is given in the paper.

Dyer and Whitman [25] discuss the vectorization of the scanline z-buffer on Convex C-1 computer. The proposed algorithm in their work basically vectorizes the shading calculations, and interpolation steps from one scanline to the next.

Weinberg [97] describes an architecture for rendering with anti-aliasing and presents simulation results. The proposed architecture is composed of series of processors, namely object processors, comparators, and filter processors to carry out rasterization, hidden-surface elimination and anti-aliasing.

The previous works summarized in this section do not cover all the previous work in parallel rendering field. Surveys of other previous works can be found in [99, 101, 8, 18].

3.3 Discussion of Previous Works

In object-space parallelism, efficient parallelization of the pixel merging phase is the most critical issue because pixel merging phase introduces overhead to the parallel execution. The approaches in [81, 56] use architectures whose processors are interconnected in a tree structure for pixel merging phase. Simulation results, which do not include communication overheads, are presented in [81] for an architecture in which processors are connected in a binary tree interconnection topology. On the other hand, a transputer based architecture with different tree interconnection topologies (binary, ternary, and unary) is utilized in [56]. The main disadvantage of the both approaches is the low processor utilization in pixel merging phase due to tree topology. The processors in the lower levels of the tree (such as processors at the leaves) have substantially less work than those in the upper levels of the tree. Another approach presented in [17] utilizes network broadcast capability for pixel merging phase. The first advantage of the work presented in the paper is that it decreases the volume of communication by injecting only pixel information for “active” pixel locations in each processor into the network. Second advantage is that the volume of communication is expected to decrease at each broadcast step since each processor, which has not yet broadcast its local pixel information, deletes the local hidden pixels. This approach is well suited to architectures with network broadcast or with shared memory because the cost of broadcast is small in these machines. However, it has two disadvantages that make it not suitable for distributed-memory machines. First of all, the communication overhead will be high since each pixel should be broadcast to each processor. In addition, another main disadvantage of the work is the low processor utilization: once the processor k broadcasts its local pixels, it waits idle until the end of pixel merging phase while processors $k + 1$, $k + 2$, ..., P do some work. An architecture with a pipelined image-composition network to perform pixel merging is presented in [62]. However, full z-buffers in each processor is injected into communication network resulting in unnecessary volume of communication. In summary, low processor utilization is one of the problems in the previous approaches. Only one previous work [53] addresses this problem by dividing the screen during pixel merging phase. Therefore, it is worth to investigate algorithms that will achieve even

load distribution and higher utilization of processors – full utilization if possible – in pixel merging phase. Load balancing in pixel merging phase is another issue that is covered in only one work [53]. However, in that work, static interleaved assignment of scanlines is utilized to achieve better load balance. Adaptive division of the screen for load balance in pixel merging computations remains as an alternative to be investigated. The communication overhead is another issue which should be considered carefully. Volume of communication can be decreased by exchanging only foremost pixels in each processor. Exchanging foremost pixels rises one important question as how to extract local foremost pixels to avoid message fragmentation in pixel merging phase. No algorithms are presented in the previous works to answer this question. Therefore, efficient algorithms to perform extraction of local foremost pixels in the local rendering phase need to be investigated.

The basic concern in image-space parallelism is how to partition the image-space so that even distribution of work load is achieved and re-distribution of primitives is minimized. There are two strategies in the previous works to partition the image-space: screen is subdivided either *non-adaptively* [36, 40, 91, 19, 99, 100, 46] or *adaptively* [65, 26, 99, 100, 76].

In non-adaptive schemes, screen is subdivided into a number of equal size subregions. This raises an important question as how many regions should there be and what should be the shape of the regions. These questions are not easily answered since they depend on the characteristics of the object database to be rendered, the algorithms that are employed for rendering, and the parallel architecture. Usually, regions are shaped as rectangular regions close to square. The advantage of the rectangular shape is the higher scalability of the algorithm. The square shape is chosen to decrease the length of the boundaries, thus to decrease number of primitives duplicated and distributed. The number of regions is kept larger than the number of processors to improve the load balance. In the previous works, two assignment strategies are utilized to assign subregions to processors. Regions are assigned either in a scattered way or dynamically on demand-driven basis. Scattered assignment has the advantage that assignment of screen subregions to processors is known a priori and static irrespective of the data. However, since scattered assignment assigns adjacent regions to different processors, it loses the coherency

in image-space and increases the duplication of polygons in the overall system. In addition, since subdivision is done irrespective of input data, it is still possible that some regions of the screen is heavily loaded and some processors may perform substantially more work than others. In demand-driven approaches, regions are assigned to processors when they become idle. Demand-driven assignment may incur a lot of communication overhead in distributed-memory multicomputers. First of all, since region assignments are not known a priori, each assignment should be broadcast to all processors so that necessary polygon data is transmitted to the corresponding processor. In addition, since many processors will inject polygons to the network for different processors or for the same processor many times it is very likely that dynamic scheme will introduce high link contention. Another disadvantage of the dynamic allocation is that adjacent regions may be assigned to different processors, which results in lose of coherency and increase in the number of primitives duplicated.

In adaptive subdivision schemes, the screen is subdivided into subregions using polygon data distribution on image-space so that each subregion has almost equal work. In these schemes, the number of subregions is less than that of non-adaptive schemes. Therefore, adaptive subdivision schemes are good alternatives to non-adaptive schemes because they are expected to decrease the communication overhead and primitive duplication by keeping the number of regions at the minimum. However, the schemes utilizing adaptive subdivision require more complicated subdivision heuristics. In the previous works that utilize adaptive subdivision [65, 26, 99, 100], a 2-dimensional coarse mesh is superimposed on the screen. This mesh is used to perform screen subdivision. Therefore, the accuracy of divisions depend on the resolution of this mesh. However, execution time of subdivision heuristic and storage space also increases by mesh resolution. In all of the previous works on adaptive subdivision, polygon counts are used and subdivision heuristics execute sequentially on a single processor. In many applications, other factors such as the projection area (in number of pixels) of the polygons, which are not considered in the previous works, also affect the work load in a region.

Chapter 4

Active Pixel Merging on Hypercubes

In this dissertation, object-space parallelism (section 3.2.1) for parallel polygon rendering on hypercube-connected multicomputers is investigated. Hypercube interconnection topology and message passing structure of the hypercube multicomputer are exploited in this work. Please refer to section 2.3 for a description of the hypercube multicomputer.

A modified scanline z-buffer algorithm is proposed for local rendering phase. The nice features of this algorithm are: It avoids message fragmentation in pixel merging phase by storing local foremost pixels in consecutive memory locations efficiently. In addition, it eliminates initialization of scanline z-buffer for each scanline on the screen. Initialization of z-buffer introduces a sequential overhead to parallel rendering.

All of the processors are utilized actively throughout this pixel merging phase by exploiting the interconnection topology of hypercube and by dividing the screen among processors. The volume of communication is decreased by only exchanging local foremost pixels in each processor after local rendering phase. We propose two schemes referred to here as *pairwise exchange* scheme and *all-to-all personalized communication* (AAPC) scheme, which are suited to the hypercube topology. Pairwise exchange scheme involves minimum number of communication steps, but it has memory-to-memory copy overhead. All-to-all personalized communication scheme eliminates this overhead by increasing the number of communication steps. Our AAPC scheme differs from 2-phase direct pixel forwarding of Lee [53]. Our algorithm is 1-phase algorithm. i.e., pixels are transmitted to destination processors in a single communication phase. Hence, our algorithm avoids the intermediate z-buffering in [53] totally.

We investigate load balancing in pixel merging phase. Two heuristics, recursive subdivision and heuristic bin packing, are proposed to achieve better load balancing in pixel merging phase. These heuristics are adaptive heuristics meaning that they utilize the distribution of foremost pixels on the screen to subdivide the screen for the pixel merging phase.

Organization of this chapter is as follows. In Sections 4.1 and 4.2, some definitions and basic algorithm are presented. Section 4.3 presents the modified scanline z-buffer algorithm for the local rendering phase. Pixel merging phase on hypercube multicomputers is described in Section 4.4 where we present several algorithms utilizing different communication strategies and embedding on hypercube. We give a comparison of these schemes based on the communication overhead incurred in each scheme. Section 4.5 presents the load balancing issue in the pixel merging phase. Two algorithms are described to divide the screen adaptively in pixel merging phase. Experimental results on an Intel's iPSC/2 hypercube multicomputer are given in Section 4.6. Some results on a Parsytec CC system recently installed in our department are presented in Section 4.7.

4.1 Some Definitions

A pixel location (x,y) on the image plane is said to be *active* if at least one pixel is generated for that location. Otherwise, it is called an *inactive* pixel location. Note that different processors may generate pixels for the same location.

A pixel is said to be a *foremost* (winning) pixel, if it is the current pixel whose z value is minimum for the active pixel location. At the end of the pixel merging operation there remains *only one* winning pixel for each active pixel location.

4.2 The Parallel Algorithm

The algorithm for object-space parallel polygon rendering on hypercube multicomputer consists of the following steps:

Step 1: Polygon information is distributed to node processors by the host processor.

In this work, the host processor distributes polygons to node processors using scattered assignment scheme. In this scheme, successive polygons in the sequence are assigned to the processors in a round-robin fashion.

Step 2: (Local rendering phase) Each processor performs geometry processing, hidden-surface removal and shading for its local polygons. In this work, hidden-surface removal is accomplished by a modified scanline z-buffer algorithm. This algorithm is presented in Section 4.3.

Step 3: (Pixel merging phase) After local z-buffering, pixels generated in each processor should be merged because more than one processor may produce a pixel for the same screen coordinate. The global z-buffering operations during the pixel merging phase can be considered as an overhead to the sequential rendering. Furthermore, each global z-buffering operation necessitates interprocessor communication. Efficient implementation of the pixel merging phase is thus a crucial factor for the performance of object-space parallel rendering. In its simplest form, pixel merging phase can be performed by exchanging pixel information for all pixel locations between processors. We will call this scheme *full z-buffer merging*. This scheme may introduce large communication overhead in pixel merging phase because pixel information for inactive pixel locations are also exchanged. This overhead can be reduced by exchanging only local foremost pixels in each processor. This scheme is referred to here as *active pixel merging*.

4.3 A Modified Scanline Z-buffer Algorithm

In distributed-memory multicomputers, transmitting all data elements in one send operation takes less time than transmitting each element in distinct steps due to setup time of each message. In order to prevent message fragmentation in active pixel merging, the local foremost pixels should be stored in consecutive memory locations. In this section, an algorithm, called *modified scanline z-buffer* algorithm, which utilizes a modified scanline z-buffer scheme to store foremost pixels in consecutive memory locations efficiently, is presented. This algorithm also avoids initialization of scanline z-buffer for each scanline

by sorting polygon spans at each scanline in increasing minimum x-intersections.

When polygons are projected onto the screen (of resolution $N \times N$), some of the scanlines intersect the edges of the projected polygons. Each pair of such intersections is called a *span*. In the first step of the algorithm, these *spans* are generated and put into the *scanline span lists*. The *scanline span lists* involve a linked list for each scanline which contains the respective polygon spans. Each span is represented by a record, which contains the intersection pair (minimum x-intersection x_{min} and maximum x-intersection x_{max}) and necessary information for z-buffering and shading. Scanline span lists are constructed by inserting the spans of the projected polygons to the appropriate scanline lists in sorted (increasing) order according to their x_{min} values. This sorting allows to perform local z-buffering without initializing the scanline array for each scanline on the screen.

In the second step, spans in the scanline lists are processed, in scanline order (y order), for local z-buffering and shading. Two local arrays are used to store only local foremost pixels. The first array is called *Winning Pixel Array* (WPA), used to store the foremost (winning) pixels. Each entry in this array contains location information, z value, and shading information about the respective local foremost pixel. Since z-buffering is done in scanline order, the pixels in WPA are in scanline order and pixels in a scanline are stored in consecutive locations. Hence, for location information, only x value of the pixel generated for location (x, y) needs to be stored in WPA. The second array, called *Modified Scanline Array* (MSA) of size N , is a modified scanline z-buffer. $MSA[x]$ gives the index in WPA of pixel generated at location x . At the beginning, each entry of the MSA is set to zero. Moreover, a “range” value is associated with each scanline. The “range” value of the current scanline is set to one plus the index of the last pixel, which is generated by the previous scanline, in WPA. The “range” value for the first scanline is set to 1. Since spans are sorted in increasing x_{min} values, if a location x in MSA has a value less than the “range” value of current scanline, it means that location x is generated by a span belonging to previous scanlines. For such locations, the generated pixels are directly stored into WPA without any comparison. Otherwise, the generated pixel is compared with the pixel pointed by the index value. This indexing scheme and sorting of spans in scanline span list avoid re-initialization of MSA at each

scanline. However, due to comparison made with “range” value, an extra comparison is introduced for each pixel generated. These extra comparison operations are reduced as follows. The sorted order of spans in the scanline span lists assures that when a span s in scanline y is rasterized, it will not generate a pixel location x which is less than x_{min} of the previous spans. The current span s is divided into two segments such that one of the segments cover the pixels generated by previous spans in the current scanline and other segment covers the pixels generated by spans of the previous scanline. Distance comparisons are made for the pixels in the first segment. The pixels generated for the second segment are stored into WPA without any distance comparisons.

4.4 Pixel Merging on Hypercube Multicomputer

This section presents pixel merging algorithms developed for a d -dimensional hypercube multicomputer with $P = 2^d$ processors. In these algorithms, each processor initially owns local foremost pixels belonging to the whole screen of size $N \times N$. Then, a global z-buffering operation is performed so that each processor gathers pixels belonging to a horizontal screen subregion of size $N \times N/P$.

The algorithms presented in this section use different inter-processor communication strategies and different interconnection topologies that can be embedded onto hypercube. The communication overhead of each algorithm is analyzed for *full z-buffer merging* and *active pixel merging*. For the analysis, it is assumed that there are $A = N \times N$ pixel locations on the screen. In addition, for active pixel merging, we assume that each processor has F foremost pixels after local z-buffering, which are distributed evenly on the image-space along y-dimension, and we also assume that at each communication step processors are perfectly load balanced. Perfect load balance and even distribution assumptions are made to simplify the analysis of each algorithm.

4.4.1 Ring Exchange Scheme

One way of performing pixel merging is to embed a ring on the d -dimensional hypercube as in Fig. 2.4(a) and perform the pixel merging on the ring. In the ring exchange scheme, each processor receives pixels from right neighbor in the ring and sends pixels to the left

neighbor. In this scheme, the screen is divided into P regions and numbered from 0 to $P - 1$. At exchange step i ($i = 1, \dots, P - 1$), processor p transmits the pixels in the region $(k + i) \bmod P$ to the left neighbor in the ring and receives the pixels in the region $(k + i + 1) \bmod P$ from the right neighbor. Here, k denotes the position of the processor in gray-code ordering [75, 79]. The receiving processor merges the pixels in the received screen region with the local region and stores them in order to transmit in the next step. These exchange operations are repeated $P - 1$ times.

For full z-buffer merging, at each communication step, A/P pixels are sent and received. The communication time in this scheme is equal to

$$T_{comm} = (P - 1)t_{su} + \frac{P - 1}{P}At_{trfull}. \quad (4.1)$$

For *active pixel merging*, at exchange step i , the processor p sends the foremost pixels to the left neighbor in the ring and receives active pixels from the right neighbor. The receiving processor merges these pixels with the local foremost pixels. The number of pixels after this merge operation is equal to the number of active pixel locations in the union of two sets: set of local active pixel locations and set of received pixel locations. If the processor has L foremost pixels and receives R pixels, then at the end of merge operation at step i , the number of foremost pixels will be $L + C_i$, where $0 \leq C_i \leq R$, assuming $R \leq L$. If two sets are totally distinct then no pixels are merged, making C_i equal to R . Therefore, the communication time in active pixel merging is equal to

$$T_{comm} = (P - 1)t_{su} + \left(\frac{P - 1}{P}F + \sum_{i=1}^{P-2} (P - i - 1)C_i \right) t_{tractive}. \quad (4.2)$$

As is seen from the equation, the volume of communication in active pixel merging depends both on the number of local foremost pixels and the distribution of pixels in the subregion for which merging is performed. In the equations above, t_{trfull} denotes the time to transmit one pixel location on z-buffer and $t_{tractive}$ denotes the time required to transmit one active pixel information. The setup time for a message is denoted by t_{su} .

4.4.2 2-dimensional Mesh Exchange Scheme

A 2-dimensional mesh with $M = 2^{\lceil d/2 \rceil}$ columns and $K = 2^{\lfloor d/2 \rfloor}$ rows can be embedded in a hypercube with $P = M \times K$ processors. Note that in mesh embedding, each row and each column of the mesh form separate rings in gray-code order. Pixel merging can be done using these rings in the mesh embedding. First, the screen is divided into M regions and the processors at each row, independently from other rows, merge these M regions in the row they belong. After these merge operations, nodes on the same column have the same screen region of size A/M pixels. Each of these screen regions are further divided into K regions, and pixel merging is done in the columns of the mesh.

The communication time required for 2-dimensional mesh exchange scheme is the sum of the communication time required for row exchange (T_{row}) and column exchange (T_{column}). That is the communication time T_{comm} is equal to

$$T_{comm} = T_{row} + T_{column}. \quad (4.3)$$

Since rows and columns are simply rings, we can use the equations for ring exchange scheme. For full z-buffer merging, A/M pixels are sent and received at each exchange stage. Therefore, communication time for the row exchanges is equal to

$$T_{row} = (M - 1)t_{su} + \frac{M - 1}{M} A t_{trfull}. \quad (4.4)$$

After the row exchanges, the screen is further divided. Hence, for full z-buffer merge, $A/(MK)$ pixels are transmitted and received. As a result, the communication time for column exchanges is equal to

$$T_{column} = (K - 1)t_{su} + \frac{K - 1}{MK} A t_{trfull}. \quad (4.5)$$

Total time of communication in 2-dimensional mesh exchange scheme for full z-buffer merging is

$$T_{comm} = T_{row} + T_{column}$$

$$\begin{aligned}
&= (M + K - 2)t_{su} + \left(\frac{M-1}{M}A + \frac{K-1}{P}A\right)t_{trfull} \\
&= (M + K - 2)t_{su} + \frac{P-1}{P}At_{trfull}.
\end{aligned} \tag{4.6}$$

Using a similar approach, communication time for row exchanges in active pixel merging is equal to

$$T_{row} = (M-1)t_{su} + \left(\frac{M-1}{M}F + \sum_{i=1}^{M-2}(M-i-1)C_i\right)t_{tractive}. \tag{4.7}$$

After the row exchanges, the remaining number of foremost pixels ($L_{foremost}$) at each processor is equal to

$$L_{foremost} = \frac{F}{M} + \sum_{i=1}^{M-1} C_i. \tag{4.8}$$

As in full z-buffer merging the remaining pixel set is further divided to exchange in the columns of the mesh. Therefore, the communication time for column exchange is equal to

$$\begin{aligned}
T_{column} &= (K-1)t_{su} + \left(\frac{K-1}{K}L_{foremost} + \sum_{i=1}^{K-2}(K-i-1)B_i\right)t_{tractive} \\
&= (K-1)t_{su} + \left(\frac{K-1}{P}F + \frac{K-1}{K}\sum_{i=1}^{M-1}C_i \right. \\
&\quad \left. + \sum_{i=1}^{K-2}(K-i-1)B_i\right)t_{tractive}.
\end{aligned} \tag{4.9}$$

As a result, total communication time (T_{comm}) is equal to

$$\begin{aligned}
T_{comm} &= (M + K - 2)t_{su} + \left(\frac{P-1}{P}F + \sum_{i=1}^{M-2}(M-i-1)C_i \right. \\
&\quad \left. + \frac{K-1}{K}\sum_{i=1}^{M-1}C_i + \sum_{i=1}^{K-2}(K-i-1)B_i\right)t_{tractive}.
\end{aligned} \tag{4.10}$$

The 2-dimensional mesh is a generalized version of ring exchange scheme since a ring can be considered as a 2-dimensional mesh with $M = P$ and $N = 1$. It is possible to embed meshes of higher dimensions onto the hypercube. In the following section, a general k -dimensional mesh exchange scheme is derived and analyzed.

4.4.3 K-dimensional Mesh Exchange Scheme

Assume we embed a k -dimensional mesh onto the hypercube as $P = 2^d = \prod_{i=0}^{d-1} L_i$. Here, L_i represents the number of processors in i^{th} dimension of the mesh with $L_i \neq 1$ for $i = 0, \dots, k-1$ and $L_i = 1$ for $i = k, \dots, d-1$. A ring is obtained by making $L_0 = P$ and $L_i = 1$ for $i = 2, \dots, d-1$. In the k -dimensional mesh, a similar exchange scheme as in 2-dimensional mesh exchange is applied. That is, pixel merging is done over the rings embedded at each dimension. At the stage i of the pixel merging in k -dimensional mesh, the rings embedded in dimension i is utilized to perform the pixel merging.

For full z-buffer merging, communication time is equal to the sum of communication times at each stage. The communication time (T_i) at stage i is equal to the communication time for pixel merging in the corresponding ring in dimension i of the k -dimensional mesh:

$$T_i = (L_i - 1)t_{su} + \frac{L_i - 1}{\prod_{j=0}^i L_j} At_{trfull}. \quad (4.11)$$

The total communication time is equal to

$$\begin{aligned} T_{comm} &= \sum_{i=0}^{k-1} T_i \\ &= \sum_{i=0}^{k-1} (L_i - 1)t_{su} + \sum_{i=0}^{k-1} \frac{L_i - 1}{\prod_{j=0}^i L_j} At_{trfull} \\ &= \sum_{i=0}^{k-1} (L_i - 1)t_{su} + \frac{P - 1}{P} At_{trfull}. \end{aligned} \quad (4.12)$$

For active pixel merging, the communication time at stage i is equal to

$$T_i = (L_i - 1)t_{su} + V_i t_{tractive} \quad (4.13)$$

where volume of communication (V_i) is equal to

$$\begin{aligned}
 V_i = & \frac{L_i - 1}{\prod_{j=0}^i L_j} F \\
 & + \sum_{j=0}^{i-1} \left[\frac{L_j - 1}{\prod_{l=j+1}^i L_l} \sum_{n=1}^{L_j-1} C_n^j \right] \\
 & + \sum_{j=1}^{L_i-2} C_j^i (L_i - j - 1).
 \end{aligned} \tag{4.14}$$

Here, C_l^j represents volume of communication incurred due to the distribution of active pixel locations in a region at the communication step l in the ring embedded in dimension j of the mesh.

The first and second terms in Eq. (4.14) represent the volume of communication incurred due to the active pixel locations in each processor before stage i . The last term in the equation represents the volume of communication incurred due to the distribution of active pixels in a region in each processor. This term also affects the volume of communication in the later stages of the pixel merging since it affects the number of active pixels in a processor after stage i . Therefore, if the volume of communication due to this term is minimized at each stage, the total volume of communication is expected to reduce. One way to minimize the value of this term is to control the distribution of active pixel locations in each region. Controlling the active pixel distribution requires a preprocessing step before the distribution of primitives to processors. This preprocessing results in redistribution of polygons between processors before local z-buffering. Note that this preprocessing step should be repeated when viewing direction and orientation changes. Another way to minimize the value of the last term in the equation is to minimize the value of L_i at each stage. The last term is minimized when $L_i = 2$ (for $i = 0, \dots, d - 1$) is chosen for the rings at each dimension and a d -dimensional mesh is embedded onto the hypercube.

Figure 4.1 illustrates volume of communication on different k -dimensional meshes on 16 processors for different scenes (see figures 4.14 – 4.16 for the rendered images of the scenes). As is seen in the figure volume of communication decreases as the dimension of the mesh increases. The lowest volume of communication is achieved on 4-dimensional

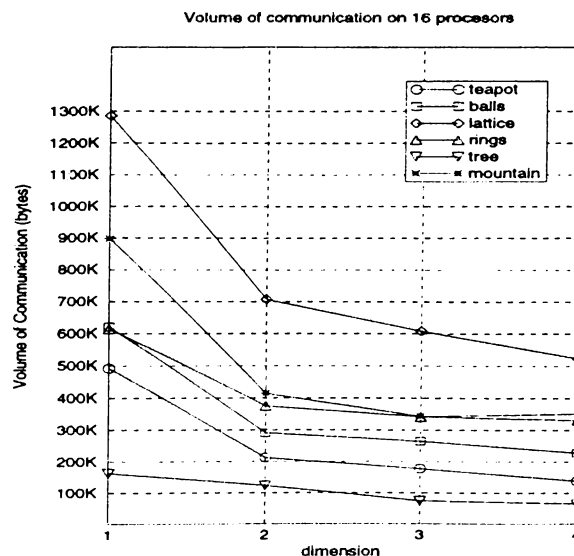


Figure 4.1: Volume of communication on different meshes embedded on the hypercube of 16 processors for different scenes.

mesh while highest is obtained on 1-dimensional mesh, i.e., ring exchange scheme. This figure supports our discussion and analysis in this section that lowest volume of communication is expected to occur when a d -dimensional mesh is embedded on a d -dimensional hypercube.

The scheme to implement pixel merging on the d -dimensional mesh (with $L_i = 2$) on hypercube is given in the next section. This scheme is called *pairwise exchange scheme*.

4.4.4 Pairwise Exchange Scheme

This scheme exploits the *recursive-halving* idea widely used in hypercube-specific global operations. This operation requires d concurrent divide-and-exchange stages. Within each stage i (for $i = 0, 1, 2, \dots, d-1$), each processor divides horizontally its current active region of size $N \times n$ into two equal sized subregions (each of size $N \times n/2$), referred here as top and bottom subregions, where $n = N$ during the initial halving stage. Meanwhile, each processor divides its current local foremost pixels into two subsets as belonging to these two subregions, which are referred here as top and bottom pixel subsets. Then, processor pairs which are neighbors over channel i exchange their top and bottom pixel subsets. After the exchange, processors concurrently perform z-buffering operations between retained and received pixel subsets to finish the stage.

For *full z-buffer merging*, at each exchange stage half of the current screen is transmitted and merged. Therefore, the total time required for inter-processor communication is equal to

$$\begin{aligned} T_{comm} &= dt_{su} + \sum_{i=0}^{d-1} \frac{A}{2^{(i+1)}} t_{trfull} \\ &= dt_{su} + \frac{P-1}{P} A t_{trfull}. \end{aligned} \quad (4.15)$$

For *active pixel merging*, at each exchange stage, each processor transmits half of the current foremost pixels. Assuming perfect load balance at each exchange step, the communication time in active pixel merging is equal to

$$T_{comm} = dt_{su} + \left(\frac{P-1}{P} F + \sum_{i=0}^{d-2} \frac{2^{(d-i-1)} - 1}{2^{(d-i-1)}} C_1^i \right) t_{tractive}. \quad (4.16)$$

4.4.5 All-to-All Personalized Communication Scheme

The schemes discussed above are also called *store-and-forward* schemes. At each exchange step, the received pixels are stored into the local memory of the processor. These pixels are compared and merged with the pixels stored before. After this merge operation, some part of the foremost pixels are sent at the next exchange step, i.e., they are forwarded towards the destination processor through other processors at each concurrent communication step. Note that during this store-compare-and-forward steps, pixels may be copied from memory of one processor to memory of the other processors more than once as is seen in the equations. This memory-to-memory copy operations can be reduced by sending the pixels directly to destination processors. This section presents a scheme called *all-to-all personalized communication* to implement this idea.

In iPSC/2 hypercube multicomputer, with DCM technology, communication between two non-neighbor processors is almost as fast as neighbor communications if all the links between two processors are not currently used by other messages. The communication hardware uses the e-cube routing algorithm [66]. Using DCMs, we can exchange messages between non-neighbor processors by the following algorithm [1]. This algorithm ensures

that at each exchange step, the pixel data is directed to destination processors with the pixel data following disjoint paths.

```

m : this node's id
Bk : pixel data belonging the partition of screen assigned to processor k.
for i = 1 to P-1 do
    k = m ⊕ i; { ⊕ represents the bitwise exclusive-or }
    send pixel data Bk to processor k;
    receive pixel data from processor k;
    sync;
endfor

```

In all-to-all personalized communication, the screen is divided into P regions. Each partition is implicitly assigned to a processor. Then, processor i sends the pixels belonging to the partition of the processor k directly to processor k . Processors, after receiving the pixels, wait for the synchronization (*sync*) so that no processor gets ahead of the others and blocks the links to be used by others. This synchronization operation can be executed in $O(d)$ time. After $P-1$ exchange steps, each processor z-buffers the local pixels and the pixels it receives from other processors. For this, each processor holds a z-buffer of size $N \times N/P$. Local pixels are scattered onto the z-buffer without any distance comparisons. Then, each received pixel's z value is compared with the z value in the pixel location in the z-buffer. After all the pixels are processed z-buffer contains the winning pixels belonging to the final picture.

For full z-buffer merging, the communication time is equal to

$$T_{comm} = (P-1)(1+d)t_{su} + \frac{P-1}{P}At_{trfull}. \quad (4.17)$$

For active pixel merging, the communication time is equal to

$$T_{comm} = (P-1)(1+d)t_{su} + \frac{P-1}{P}Ft_{tractive}. \quad (4.18)$$

4.4.6 Comparison of Pixel Merging Schemes

As is seen from the equations, the volume of communication in full z-buffer merging is not affected by distribution of foremost pixels in screen regions. The volume of communication in active pixel merging in all of the *store-and-forward* schemes are affected by

the distribution of pixels in a region in active pixel merging, *pairwise exchange* scheme being the least affected. The volume of communication in *all-to-all personalized communication* scheme, on the other hand, is not affected by the distribution of pixels. Hence, among all schemes, all-to-all personalized communication scheme is expected to give the lowest volume of communication in active pixel merging. For large number of processors with high communication latency, the number of steps, which directly affects the total setup time, in pixel merging phase is also a crucial factor. In pairwise exchange scheme, the number of communication steps increases with d , being the smallest among store-and-forward schemes, whereas it increases in $O(P \log_2 P)$ in all-to-all personalized communication scheme. For large number of processors, the number of steps may be a dominating factor in communication time in active pixel merging phase. Therefore, among all schemes presented in this section, *pairwise exchange scheme* and *all-to-all personalized communication scheme* are most suitable for pixel merging on hypercube multicomputers. Only these two schemes are experimentally investigated in this work.

4.5 Load Balancing in Pixel Merging Phase

In this section, two heuristics that implement adaptive subdivision of screen among processors to achieve good load balance in pixel merging are presented.

4.5.1 Recursive Adaptive Subdivision

This scheme recursively divides the screen into two subregions such that number of pixels in one subregion is almost equal to the number of pixels in the other subregion. This scheme is well suited to the recursive structure of the hypercube and can be done in parallel.

Each processor counts the number of local foremost pixels at each scanline and stores them in an array. Each entry of the array stores the sum of local foremost pixels at the corresponding scanline. An element-by-element global sum operation is performed on this array to obtain the distribution of foremost pixels in all processors. Then, using this array, each processor divides the screen into two horizontal bands of consecutive scanlines so that each region contains an equal number of active pixel locations. Along

with the division of the screen, the hypercube is also divided into two equal subcubes of dimension $d - 1$. Top subregion is assigned to one subcube while bottom subregion is assigned to the other subcube. Subcubes perform subdivision of the local subregions concurrently and independently. Since the screen is divided into horizontal bands, the global array obtained by global sum operation is used for further divisions of the screen.

This algorithm needs an extra step for all-to-all personalized communication scheme. In all-to-all personalized communication scheme, each processor requires the information on screen subregions assigned to other processors. In order to gather this information on all nodes, a global collect operation is performed on the screen subregions assigned to each processor after the subdivision of the screen is done.

4.5.2 Heuristic Bin Packing

In the recursive adaptive subdivision scheme, the subdivision of the screen is done on scanline basis, i.e., scanlines are not divided. For this reason, it is difficult to achieve exactly equal load in each subregion. In addition, when a division point is found and the screen is divided into two subregions, each subregion is subdivided independent of the other one. As a result, at each recursive subdivision, the load imbalance between the subregions may propagate and increase. At the end of recursive subdivision, some processors may still have substantially more work load than others. A better distribution of work load among the processors can be achieved by using a different partitioning scheme, called *heuristic bin packing*. In this scheme, the goal is to minimize the difference between the loads of the maximum loaded processor and minimum loaded processor. In order to realize this goal, a scanline is assigned to a processor with minimum work load. In addition, scanlines are assigned in decreasing number of pixels they have, i.e., scanlines that have large number of pixels are assigned at the beginning. In this way, large variations in the processor loads due to new assignments are minimized towards the end.

In each processor, the total number of pixels at each scanline after local hidden surface removal step is found. Then, scanlines are sorted with respect to number of pixels in decreasing order. This sorting is done in parallel. Assume that the size of the set of scanlines, which have non-zero number of pixels, is S . For parallel sorting, each processor

sorts a disjoint subset of size S/P of this set of scanlines in parallel. Then, sorted arrays in each processor are merged to obtain the final sorted array. This merge operation can be performed in d concurrent communication steps. In this work, load balancing in parallel sorting operation is not considered. Various parallel sorting algorithms can be found in [1, 71]. In this scheme, the minimum work loaded processor to assign the scanline is found using a binary heap.

During local hidden surface removal, the foremost pixels are stored into WPA in scanline order in consecutive locations. However, the load balancing algorithm may assign consecutive scanlines to different processors. Hence, non-consecutive scanline data in the winning pixel array of the processor l can be assigned to the processor k . As a result, in order processor l to send the pixels belonging to scanlines assigned to processor k , it has to gather those pixels in another array so that they are stored in consecutive memory locations. In order to avoid this extra gather operation, the load balancing algorithm is executed before local hidden surface removal and scanlines are renumbered so that scanlines assigned to a processor are numbered consecutively. In this way, pixels generated for these scanlines are stored in consecutive locations in winning pixel array. However, the load metric in heuristic bin packing algorithm is the number of pixels in each scanline after local hidden surface removal is performed. In order to find the number of winning pixels after local hidden surface removal without running local z-buffer operations, each processor executes the algorithm called *extended span algorithm* given in Fig. 4.2 on spans in the *span list* structure.

In this algorithm, intersecting spans in scanline y are merged to form extended spans. The number of pixels in these extended spans gives the number of winning pixels after local z-buffering for scanline y . Remember that during scanline span list creation, spans are sorted with respect to their x_l values in increasing order. Because of the sorting, there is no need to store the extended spans. In addition, checking the intersection of a span s with the extended span can be done by only checking x_l of span s with $\text{extnd_span_}x_r$.

```

Initialize ScanPixelCount array to zero.
for ( each scanline  $y$  ) do
    extnd_span_ $x_r$  = -1;
    extnd_span_ $x_l$  = 0;
    for ( each span  $s$  in scanline  $y$  ) do
        if (  $x_l < \text{extnd\_span\_}x_r$  ) then
            if (  $x_r > \text{extnd\_span\_}x_r$  ) then
                extnd_span_ $x_r$  =  $x_r$ ;
            endif
        else
            ScanPixelCount[i] = extnd_span_ $x_r$  - extnd_span_ $x_l$  + 1;
            extnd_span_ $x_r$  =  $x_r$ ;
            extnd_span_ $x_l$  =  $x_l$ ;
        endif
    endfor
    ScanPixelCount[i] = extnd_span_ $x_r$  - extnd_span_ $x_l$  + 1;
endfor

```

Figure 4.2: Extended span algorithm.

4.6 Experimental Results on an iPSC/2 Hypercube Multicomputer

The algorithms proposed in this work were implemented in C language on a 16-node Intel iPSC/2 hypercube multicomputer. Algorithms were tested for scenes composed of 1, 2, 4, and 8 tea pots for screens of size 400×400 , 640×640 , and 800×800 . The abbreviations in the figures and tables are AAPC: *all-to-all personalized communication scheme*, PAIR: *pairwise exchange scheme*, RS: *recursive subdivision scheme*, HBP: *heuristic bin packing scheme*, ZBUF-EXC: *full z-buffer merging*. All timing results in the tables are in milliseconds.

Table 4.1 gives the characteristics of the scenes in terms of total number of pixels generated, number of polygons and total number of winning pixels in the final picture for different screen sizes. Rendered images of the scenes from the viewing directions used in the experiments are given in figures 4.11 – 4.13.

Table 4.1: Scene characteristics in terms of total number of pixels generated (TPG), number of triangles, and total number of winning pixels in the final picture (TPF) for different screen sizes.

Scene	Num. Of Triangles	N=400		N=640	
		TPG	TPF	TPG	TPF
1 POT	3751	59091	43247	137043	110515
2 POT	7502	66802	37084	151881	94840
4 POT_1	15004	71578	26328	146468	66727
4 POT_2	15004	81735	35629	171480	90692
8 POT_1	30008	154187	52258	324464	133617
8 POT_2	30008	99589	36043	201829	91729

Table 4.2 illustrates the performance comparison of PAIR-RS scheme with full z-buffer merging. The timings for some scene instances for ZBUF-EXC scheme could not be obtained due to insufficient local memory. Those cases are indicated by a “*” in this table. As seen in Table 4.2, PAIR-RS gives much better results than ZBUF-EXC in pixel merging phase. Since pixel information for inactive pixel locations are also exchanged, the volume of communication in ZBUF-EXC is larger than that of PAIR-RS. As is also seen from the table, the PAIR-RS performs better than ZBUF-EXC also in local z-buffer phase since it avoids initialization of z-buffer.

Table 4.3 illustrates the performance comparison of AAPC-HBP, AAPC-RS, and PAIR-RS schemes. The timing results for *local z-buffer* do not include the time spent on span list creation, because all algorithms use the same span list creation algorithm. The overheads associated with load balancing operations are incorporated into *local z-buffer* operation. If we compare the pixel merging times, AAPC-HBP scheme gives the best results among all schemes. This is because of the fact that the *heuristic bin packing* scheme achieves better load balancing than *recursive adaptive subdivision* scheme. As is also seen from the table, PAIR-RS scheme gives worst performance results in pixel merging phase. This is because of the store-and-forward overhead associated with this scheme. If performance of the algorithms are compared with respect to execution time of

Table 4.2: Relative execution times (in milliseconds) of full z-buffer merging and PAIR-RS for N=400.

P	Scene	PAIR-RS			ZBUF-EXC		
		Span List Creation	Local z-buffer	Pixel Merging	Span List Creation	Local z-buffer	Pixel Merging
16	1 POT	322	434	348	316	578	2015
	2 POT	481	471	341	470	585	1940
	4 POT.1	1038	520	323	1015	647	1930
	4 POT.2	1124	579	408	1099	702	1958
	8 POT.1	2142	1079	684	2104	1128	2043
	8 POT.2	2087	701	451	2029	805	1958
8	1 POT	630	815	468	612	952	1941
	2 POT	947	886	475	920	989	1882
	4 POT.1	2037	989	419	1968	1093	1798
	4 POT.2	2268	1109	545	2186	1191	1881
	8 POT.1	4219	2030	861	*	*	*

local z-buffer operation, algorithms that use *recursive adaptive subdivision* scheme perform better. This is due to the fact that *recursive adaptive subdivision* scheme introduces less overhead to the execution. In *Total* (local z-buffer + pixel merge) execution time, AAPC-HBP scheme achieves best performance for all instances.

Performance comparison of load balancing heuristics is done in Fig. 4.3. The *load imbalance* is the ratio of the difference of the work loads of maximum and minimum loaded processors to average work load. The work load of a processor was taken to be the number of pixel merging operations it performs in the pixel merging phase. As seen from the figure, heuristic bin packing achieves much better load balance than recursive adaptive subdivision as expected. Load balance improves with increasing screen resolution due to better accuracy in dividing the screen. As is also seen from Fig. 4.3(a), heuristic bin packing scales better than recursive subdivision for larger number of processors.

Total volume of concurrent communication (in bytes) for various pixel merging schemes are illustrated in Fig. 4.4. The total volume of concurrent communication is calculated as the sum of the maximum volume of communication at each communication step. As seen from the figure, *all-to-all personalized communication* scheme results in less volume of communication than *pairwise exchange* scheme as expected. Note that the volume of communication in active pixel merging is proportional to the number of active pixel

Table 4.3: Comparison of execution times (in milliseconds) of several pixel merging schemes.

N	P	Scene	AAPC-HBP			AAPC-RS			PAIR-RS		
			Local z-buff.	Pixel Merg.	Total	Local z-buff.	Pixel Merg.	Total	Local z-buff.	Pixel Merg.	Total
400	16	4 POT_1	550	181	731	524	218	742	520	323	843
		8 POT_1	1126	302	1428	1083	376	1459	1079	684	1763
	8	4 POT_1	1031	250	1281	992	291	1283	989	419	1408
		8 POT_1	2098	464	2562	2034	543	2577	2030	861	2891
640	16	4 POT_1	1060	333	1393	1016	418	1434	1011	702	1713
		8 POT_1	2238	611	2849	2170	794	2964	2165	1502	3667
	8	4 POT_1	2013	540	2553	1951	636	2587	1947	936	2883
		8 POT_1	4250	1050	5300	4146	1242	5388	4142	1957	6099

locations in each processor. As the number of processors increases, the number of active pixel locations per processor is expected to decrease. Hence, it is expected that volume of communication decreases as the number of processors increases as is also seen in Fig. 4.4(a). The increase in volume of communication in PAIR-RS scheme on 4 processors is due to store-and-forward overheads. It is also experimentally observed that better load balance in pixel merging indirectly affects the volume of communication as well. As illustrated in Fig. 4.4(b), *heuristic bin packing* results in less volume of communication than *recursive adaptive subdivision*.

Speedup curves for different schemes are illustrated in figures 4.5 – 4.6. Due to insufficient local memory in node processors, speedup figures could only be obtained for screen sizes 400×400 and 640×640 for 1 POT and 2 POT scenes and speedup figures for ZBUF-EXC scheme could only be obtained for 400×400 screen. Figures represent the speedup curves for total execution times (span list creation + local z-buffering + pixel merging). As is seen from figures, AAPC-HBP scheme achieves higher speedup than other schemes because of less volume of communication, less number of global z-buffering operations and better load balancing in the pixel merging phase. Among all the schemes, the ZBUF-EXC scheme gives worst speedup results. This is because of the unnecessarily large volume of communication and large number of global z-buffering operations in pixel merging phase.

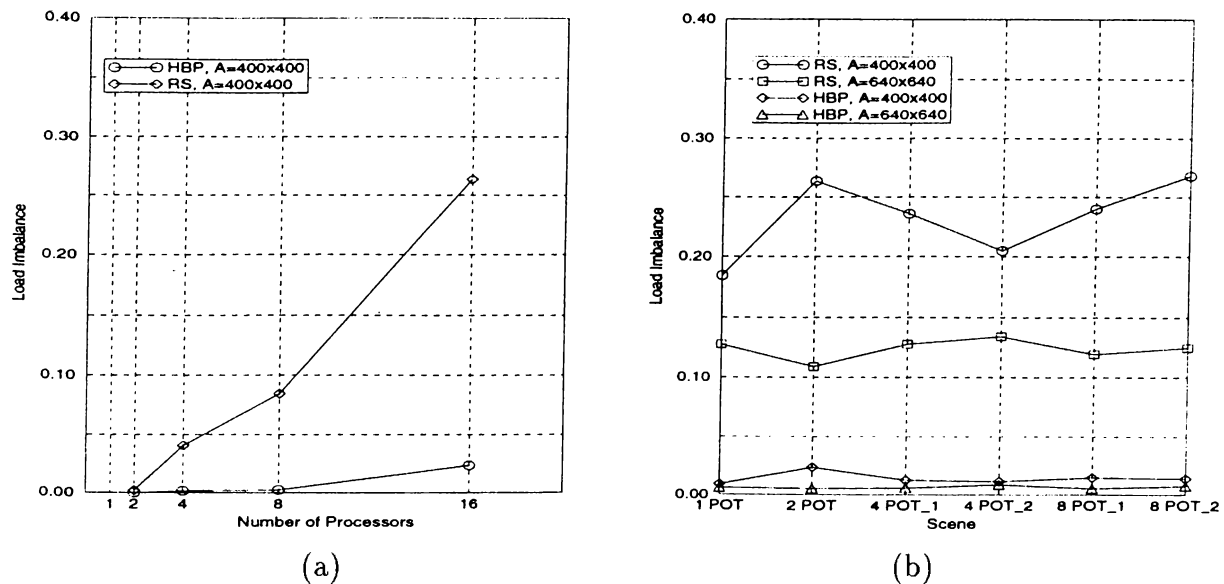


Figure 4.3: Comparison of RS with HBP. (a) Different number of processors for 2 POT scene, $A = 400 \times 400$. (b) Different screen resolutions and different scenes on 16 processors.

4.7 Results on a Parsytec CC System

This section presents the preliminary experimental results on the Parsytec CC system recently installed in our department. Each processing node of the CC system has 64 Mbytes of memory and each I/O node has 128 Mbytes of memory. The I/O nodes can also be used as processing nodes. However, unlike processing nodes, they are connected to hard disks used to store user files etc. Each node has PowerPC 604 processor running at 133 Mhz. The interconnection topology of the CC system installed in our department is shown in Fig. 4.7. Message passing between any two nodes is done through the multistage switch network using routers.

The pixel merging schemes AAPC-HBP and ZBUF-EXC were coded on the Parsytec CC system for the experiments. In these experiments, we assume a hypercube interconnection topology on the Parsytec system. The algorithms were coded in C language and PVM 3.3 [30, 72] was used for message passing. The algorithms were tested on 6 scenes from the publicly available SPD database [37]. The number of triangles in these scenes range from 102K to 524K. Table 4.4 gives the number of triangles in each scene. All results presented in this section are the timings for rendering the images in

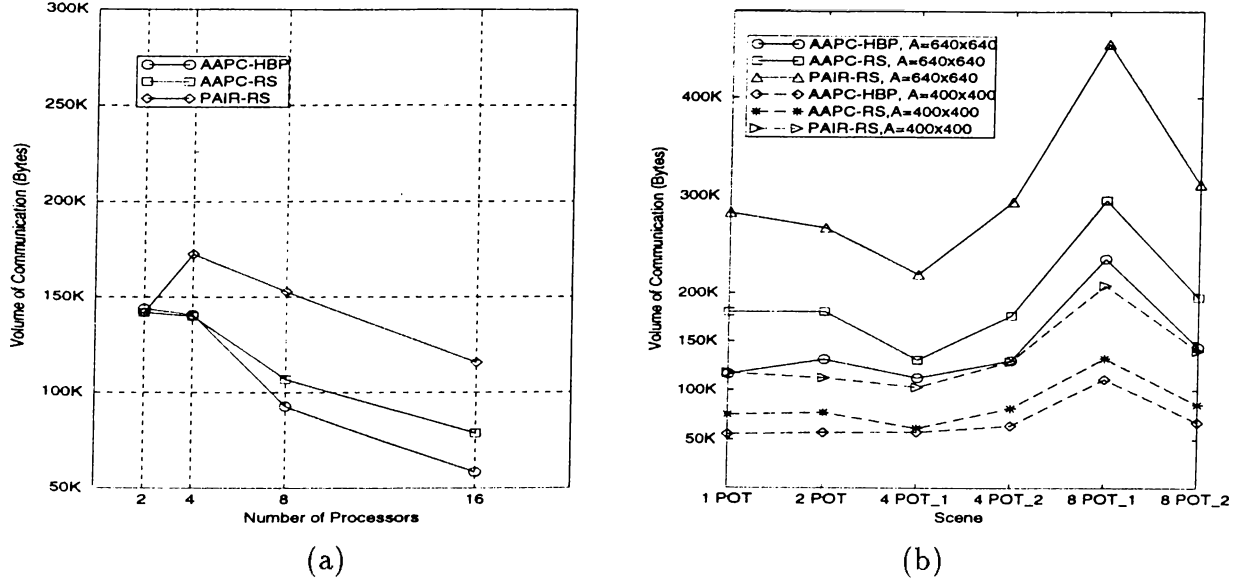


Figure 4.4: Volume of communication for (a) 2 POT scene on different processors, $A = 400 \times 400$. (b) $A = 400 \times 400$ and $A = 640 \times 640$ for different scenes on 16 processors.

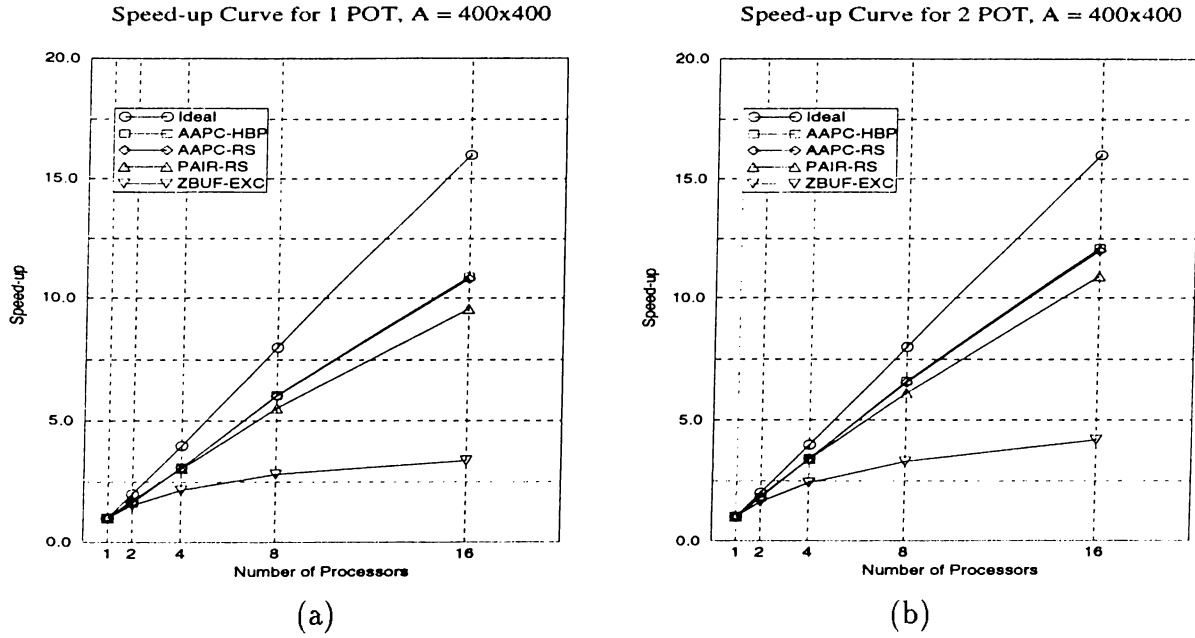
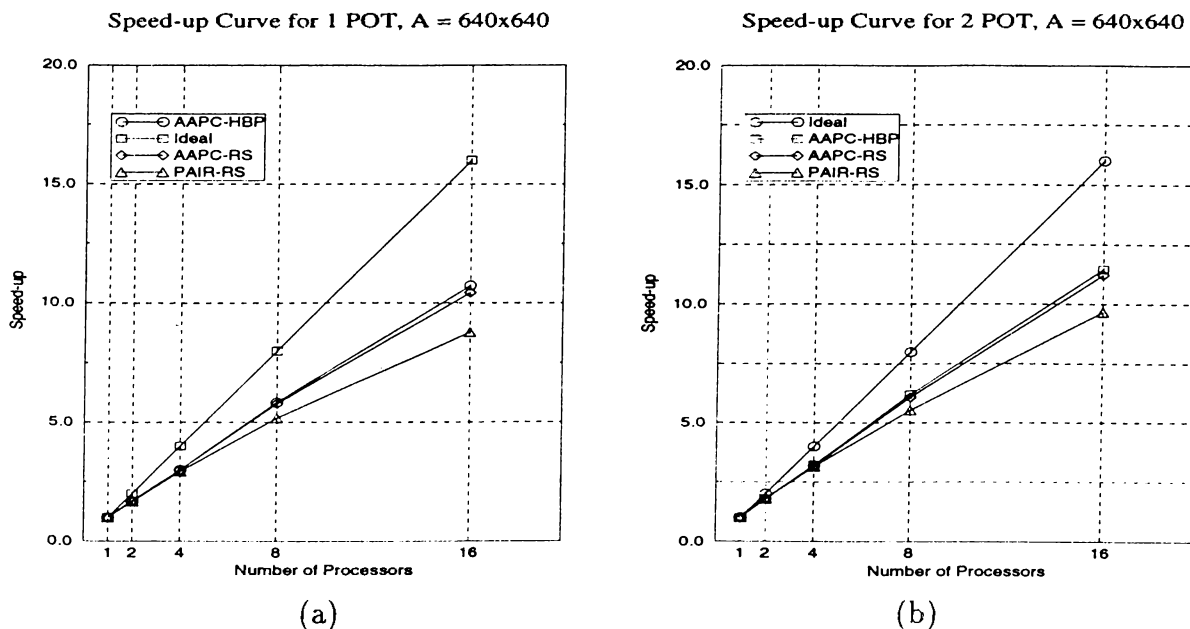


Figure 4.5: Speedup figures for $A = 400 \times 400$. (a) 1 POT scene (b) 2 POT scene.

Figure 4.6: Speedup figures for $A = 640 \times 640$. (a) 1 POT scene (b) 2 POT scene.

figures 4.14 – 4.16 at the screen resolution of 512×512 .

Table 4.4: Number of triangles in the test scenes.

Scene Description	Number of Triangles	
Teapot	102080	102K
Balls	157440	157K
Lattice	235200	235K
Rings	343200	343K
Tree	425776	426K
Mountain	524288	524K

Figure 4.8 illustrates the rendering rate of AAPC-HBP and ZBUF-EXC scheme in terms of number of triangles per second. Figure 4.9 illustrates the speedup achieved by AAPC-HBP and ZBUF-EXC algorithms. The AAPC-HBP scheme achieves rendering rate of 300K – 700K triangles per second on 16 processors. However, ZBUF-EXC scheme can achieve much lower rendering rates of 100K – 350K triangles per second. This verifies that exchanging only active pixels result in considerable gain in rendering rate of the object-space parallelism. As is seen in figure 4.9, AAPC-HBP achieves better speedup than ZBUF-EXC scheme. The AAPC-HBP scheme achieves speedup of 5–10

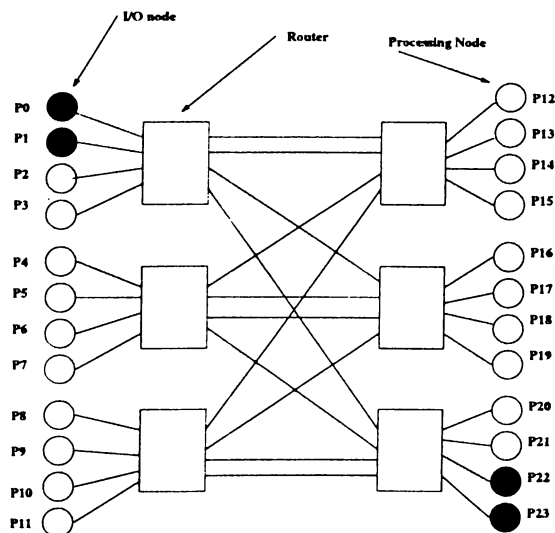


Figure 4.7: The Parsytec CC system.

while ZBUF-EXC achieves speedup values 2-6 on 16 processors. The speedup values on Parsytec CC system are lower than those on iPSC/2 system. There are various factors for the lower speedup values. One main factor is that PowerPC processors used in Parsytec system are much more powerful than 80386/387 processors of iPSC/2 hypercube. A PowerPC processor has peak performance of 266 MFlops while 80386/387 processor in iPSC/2 has peak performance of 300 KFlops. While PowerPC processors are approximately 1000 times faster than 80386/387, the peak communication bandwidth between two nodes of Parsytec CC system (40 MBytes/sec) is approximately 14 times faster than that of iPSC/2 (2.8 Mbytes/sec). Hence, interprocessor communication affects the performance of the algorithm more in Parsytec system than it affects in iPSC/2 system. In addition, PVM, which is slower than native message passing library of Parsytec system, was used for message passing in the current implementations. Another important factor is that the interconnection topology of the Parsytec CC system is not hypercube. Our algorithms exploit the connection topology of the hypercubes and the interprocessor communication structure of our current implementations on Parsytec CC system assumes a hypercube topology. It is likely that during message passing some contention for some links will incur, resulting in serialization of messages in the system.

We note that the volume of concurrent communication decreases with increasing number of processors since each processor injects less number of pixels into communication network. However, total volume of communication increases with increasing number of processors since more processors transmit pixels through the interconnection network. This situation is illustrated in Fig. 4.10. The values are the average of communication volume of all test scenes.

4.8 Conclusions

In this work, efficient algorithms were proposed for active pixel merging on hypercube multicomputers. The algorithms proposed in this chapter reduce the volume of communication by exchanging only active pixel locations in pixel merging phase. The message fragmentation in active pixel merging is avoided by storing local foremost pixels to consecutive memory locations in local z-buffering phase. An efficient algorithm, called modified scanline z-buffer, is proposed to store the local foremost pixels into consecutive memory locations efficiently. This algorithm also avoids initialization of scanline z-buffer for each scanline on the screen.

It is experimentally observed that active pixel merging with modified scanline z-buffer algorithm performs better than full z-buffer merging. It is also experimentally observed that *all-to-all personalized communication* scheme achieves less communication overhead than *pairwise exchange* scheme due to less store-and-forward overheads in active pixel merging.

Two load balancing heuristics were proposed to distribute load evenly in pixel merging. The *heuristic bin packing* achieves better load balance and scales better than *recursive adaptive subdivision* in active pixel merging. Therefore, it is recommended that *all-to-all personalized communication* with *heuristic bin packing* scheme should be utilized for active pixel merging on hypercube multicomputers.

Note that the modified scanline z-buffer algorithm and load balancing heuristics proposed in this work are independent of the interconnection topology. Hence, the algorithm and heuristics can be used, without any modification, in distributed-memory multicomputers with an interconnection topology other than hypercube. The only restriction

with recursive adaptive subdivision is that the number of processors needs to be a power of two. However, this restriction can be relaxed by dividing the screen into two parts at each step so that the ratio of work loads in each part is equal to the ratio of the number of processors in those parts. As in hypercube topology, exchanging foremost pixels is expected to give higher rendering rates than merging full z-buffers in pixel merging phase on other topologies due to much less volume of communication. However, the message exchange sequence of pixel merging schemes should be modified to avoid link contention in the target architecture to get maximum performance. The all-to-all personalized communication scheme is expected to achieve better performance than store-and-forward schemes (e.g., pairwise exchange) for many interconnection topologies since it has less memory-to-memory copy overheads. For example, 2-phase direct pixel forwarding scheme of Lee *et al.* [53] achieves better performance on 2D meshes than their store-and-forward schemes.

In this thesis, a preliminary implementation of all-to-all personalized communication with heuristic bin packing was done for a Parsytec CC system. This implementation achieves rendering rates of 300K – 700K triangles per second on 16 processors using data sets from SPD database [37]. Our preliminary implementation assumes hypercube topology and uses PVM for message passing. Thus, it is expected to achieve higher rendering rates with an implementation suited to interconnection structure of Parsytec and using native message passing library.

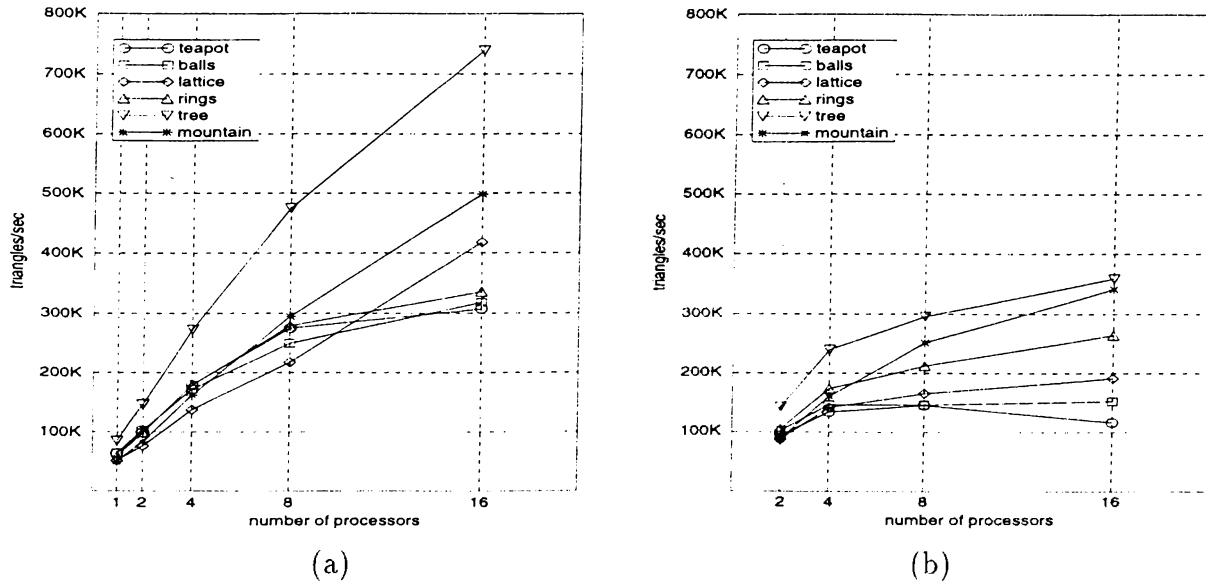


Figure 4.8: Rendering rates of algorithms on Parsytec CC system. (a) AAPC-HBP (b) ZBUF-EXC.

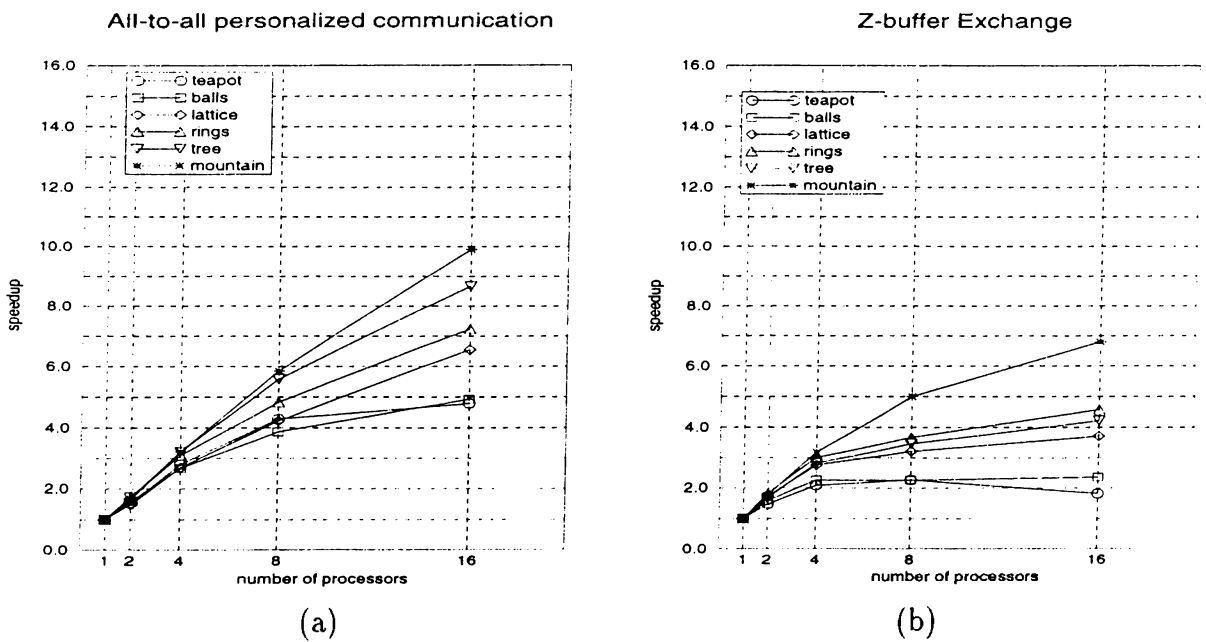


Figure 4.9: Speedup values achieved by the algorithms on Parsytec CC system. (a) AAPC-HBP (b) ZBUF-EXC.

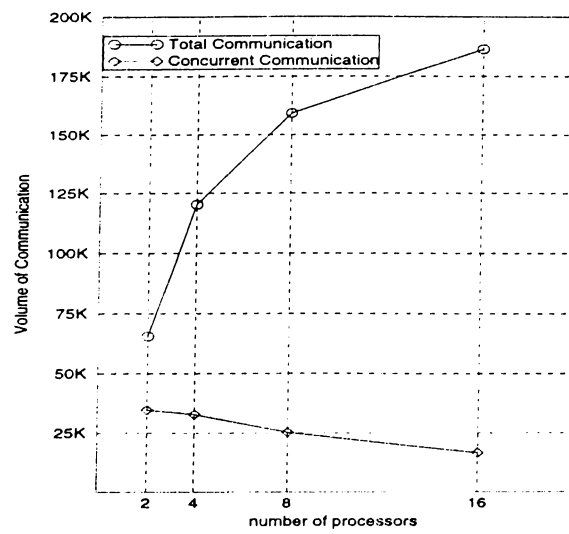
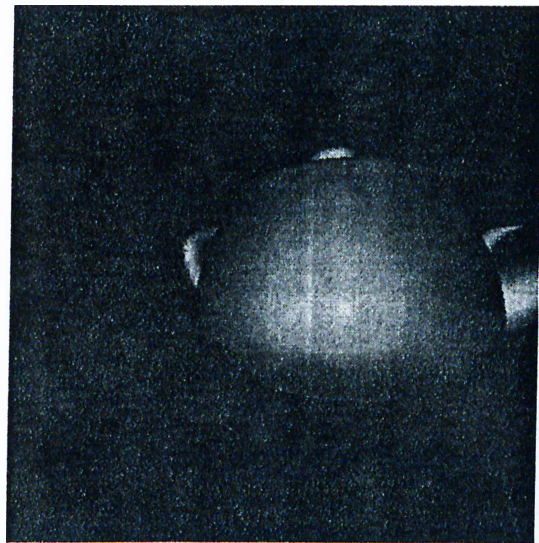


Figure 4.10: Total volume of communication and concurrent volume of communication.

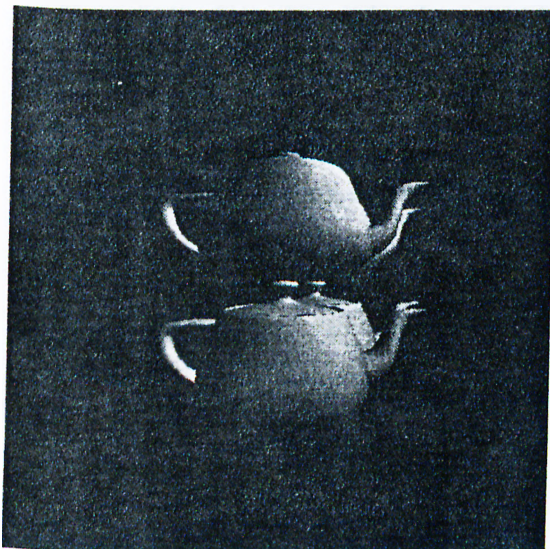


(a)



(b)

Figure 4.11: Rendered images of the scenes used in the experiments on iPSC/2. (a) 1 POT scene (b) 2 POT scene.



(a)



(b)

Figure 4.12: Rendered images of the scenes used in the experiments on iPSC/2. (a) 4 POT_1 scene (b) 4 POT_2 scene.



(a)

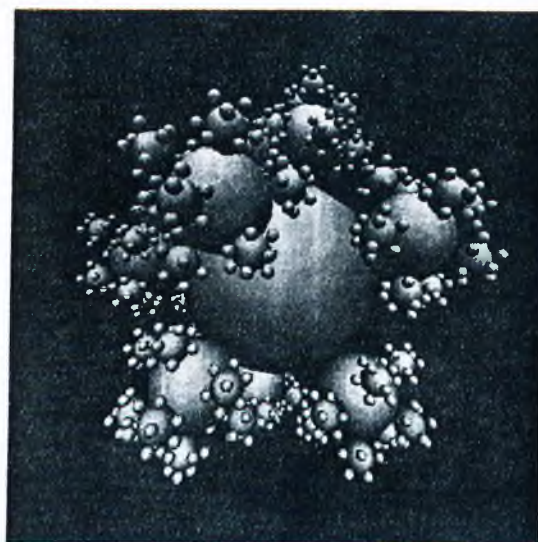


(b)

Figure 4.13: Rendered images of the scenes used in the experiments on iPSC/2. (a) 8 POT_1 scene (b) 8 POT_2 scene.

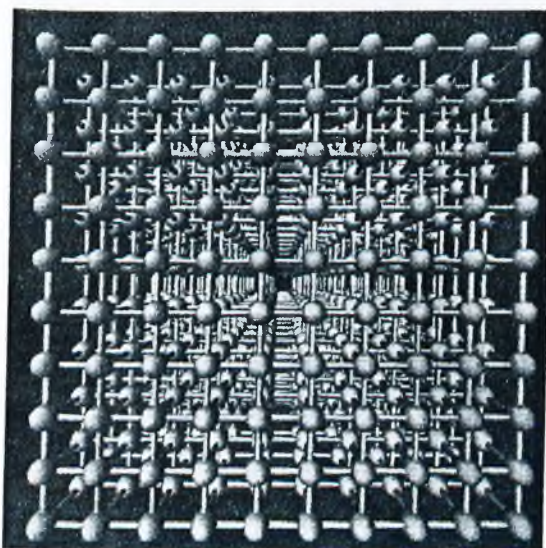


(a)

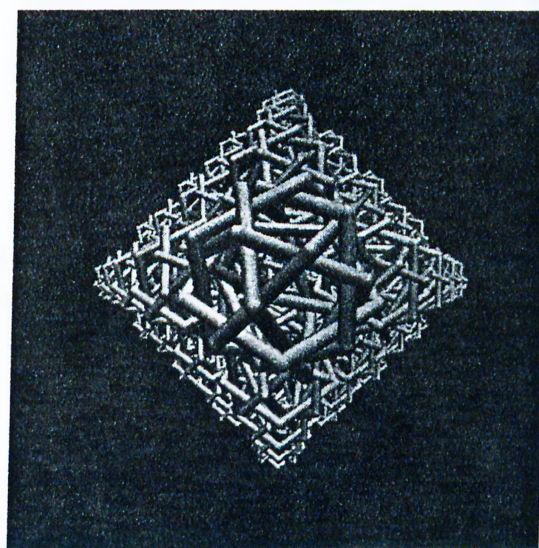


(b)

Figure 4.14: Rendered images of the scenes used in the experiments on the Parsytec CC system. (a) Teapot scene (102080 triangles, rendering time is 0.332 seconds on 16 processors) (b) Balls scene (157440 triangles, rendering time is 0.495 seconds on 16 processors).

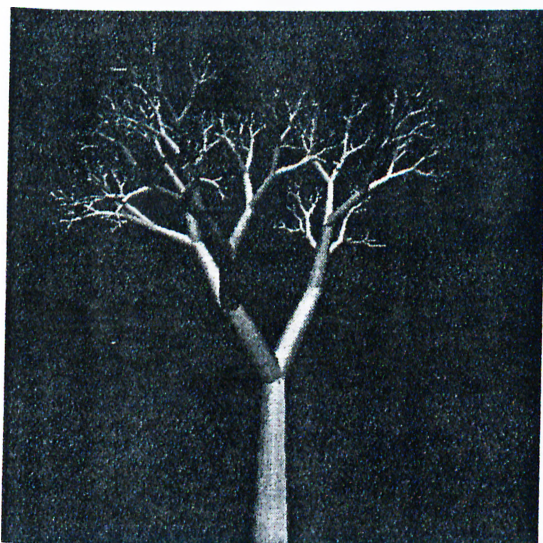


(a)

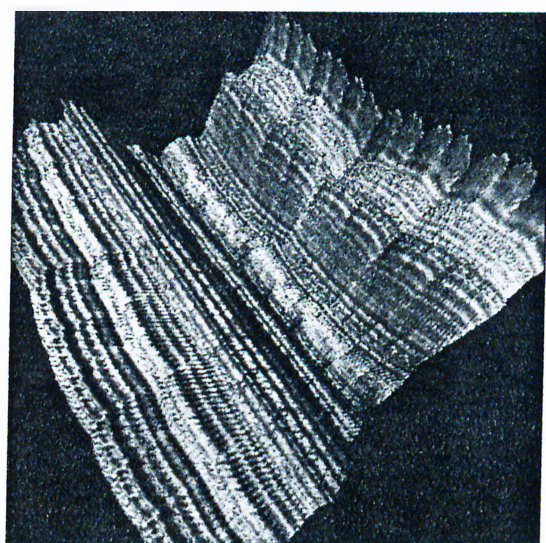


(b)

Figure 4.15: Rendered images of the scenes used in the experiments on the Parsytec CC system. (a) Lattice scene (235200 triangles, rendering time is 0.7 seconds on 16 processors) (b) Rings scene (343200 triangles, rendering time is 0.821 seconds on 16 processors).



(a)



(b)

Figure 4.16: Rendered images of the scenes used in the experiments on the Parsytec CC system. (a) Tree scene (425776 triangles, rendering time is 0.576 seconds on 16 processors) (b) Mountain scene (524288 triangles, rendering time is 1.052 seconds on 16 processors).

Chapter 5

Volume Rendering: Overview and Related Work

In many fields of science and engineering, computer simulations provide a cheap and controlled way of investigating physical phenomena. The output of these simulations is usually large amount of numerical values. Vast amounts of numerical data are also obtained by scanning physical entities by advanced scan devices. In medical imaging, for example, a specific part of human body is scanned by advanced scan devices using techniques such as magnetic resonance imaging (MRI). The outcome of the scan operation is large amounts of numerical data representing the properties of different tissues in that part of the human body. The large quantity of data makes it very difficult for the scientist and researcher to extract useful information from the data to derive some conclusions. Therefore, visualizing large quantities of numerical data as an image provides an indispensable tool for researchers. In many engineering simulations and in medical imaging, data sets consist of numerical values which are obtained at points (sample points), with 3-dimensional coordinates, distributed in a volume that represents the physical entity or the physical environment. The sample points constitute a *volumetric grid* superimposed on the volume. The process of visualizing such grids is called *volume visualization* [48], referred to here as *volume rendering*.

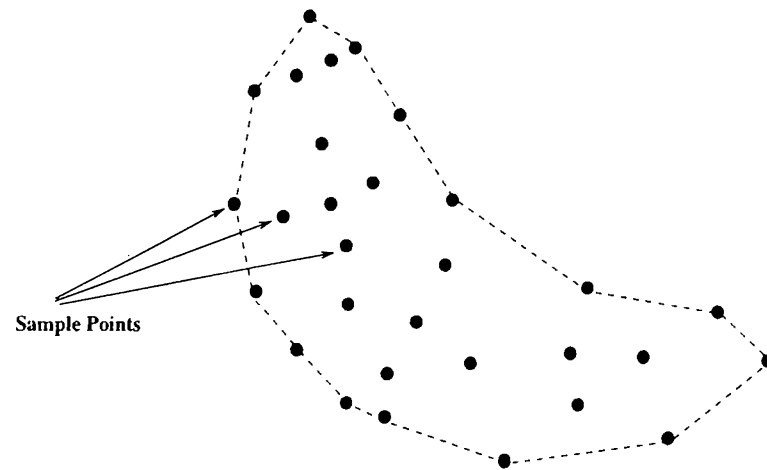
In this chapter, an overview of ray-casting based direct volume rendering of unstructured grids is given. There are two important problems the direct volume rendering algorithms for unstructured grids have to solve; *point location* and *view sort* problems.

These problems are introduced in the following sections and approaches to solve point location and view sort problems are presented. Previous works on parallel volume rendering of unstructured grids are summarized in the last section.

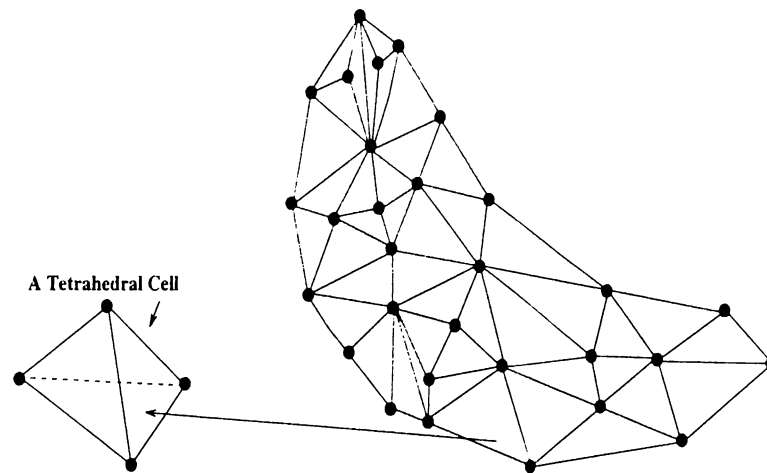
5.1 Nomenclature

A data set is called *volumetric data set* or *volume data* if data points of the set are defined in 3-dimensional space in a volume. The term *volume rendering* is used to refer to the process of visualizing volumetric data sets. The term *sample point* is used to refer to a point with 3-dimensional spatial coordinates for which a numerical value is associated. Sample points in the volume data are connected in a predetermined way to form *volume elements*, also referred to here as *cells*. Sample points that form a cell are called *vertices* of the cell. There are various cell shapes; rectangular prism, hexahedra, tetrahedra and polyhedra being the most common ones. Figures 5.1(a) and 5.1(b) illustrate a typical volume with sample points connected to form tetrahedral cells. In volumetric data sets, two or more cells may share a face. Therefore, there may exist a connectivity relation between cells. If a face of a cell is shared by two or more cells, that face is called *interior face*. If it is not shared by any other cell, the face is called *exterior face*. A cell with at least one exterior face is called *exterior cell* or *boundary cell*. Otherwise, it is called *interior cell*.

In a volumetric data set, the sample points constitute a *volumetric grid* superimposed on the volume. Therefore, the type of the grid also defines the spatial characteristics of the volumetric data set, which are important in the visualization process. There are various classifications for volumetric grids in literature [11, 84, 106, 28, 105, 90]. In this work, volumetric grids are classified into two main categories as *structured* and *unstructured* grids. Figure 5.2, based on the illustration by Yagel [106], illustrates types of grids that are commonly encountered in volume rendering. The common characteristic of the structured grids is that sample points are distributed regularly in 3-dimensional space. The distance between sample points may be constant or variable. Although this type of distribution is obvious in *cartesian*, *regular*, and *rectilinear* grids, this situation is not so obvious in *curvilinear* grids. In curvilinear grids, sample points are distributed



(a)



(b)

Figure 5.1: A volumetric data set. Figure illustrates a 2-dimensional projection of the volume. (a) Volume is sampled at 3-dimensional space. Each small filled circle represents the sample points with 3-dimensional spatial coordinates. Dashed lines represent the boundaries of the volume. (b) Sample points are connected to form volume elements. A tetrahedral cell, which is formed by connecting four distinct sample points, is also illustrated.

in such a way that the grid fits onto a curvature in space. Hence, there exists a regularity in the distribution of sample points and this type of grids are also categorized as structured grids. The cell shapes in structured grids are hexahedral cells formed by eight sample points. These type of grids are also called *array oriented* grids since these grids are usually represented as a 3-dimensional array, for which there exists a one-to-one correspondence between array entries and sample points. Due to array oriented nature of structured grids, the connectivity relation between cells are provided implicitly. In unstructured grids, on the other hand, sample points in the volume data are distributed irregularly over three dimensional space and there may be voids in the volumetric grid. The spacing between sample points is variable. There exists no constraint on the cell shapes. Common cell shapes are tetrahedra and hexahedra shapes. Unstructured grids are common in engineering simulations such as computational fluid dynamics (CFD), finite volume analysis (FVA) simulations, and finite element methods (FEM). In addition, *curvilinear* grid types are also common in CFD. Unstructured grids are also called *cell oriented* grids. They are represented as a list of cells with pointers to sample points that form the respective cells. Due to cell oriented nature and irregular distribution of sample points, the connectivity information between cells are provided explicitly if it exists. In some applications, simulations do not require a connectivity information. In such cases, the connectivity between cells may not be provided at all. Unstructured grids can further be divided into three subtypes as *regular*, in which cell shapes are consistent and usually tetrahedral cells with at most two cells sharing a face, *irregular*, in which there is not consistency in cell shapes and a face may be shared by more than two cells, and *hybrid*, which is the combination of structured and unstructured grids.

In this work, the term *direct volume rendering* (DVR) refers to the process of visualizing the volume data without generating an intermediate geometrical representation such as isosurfaces. Other techniques, called *surface rendering* [48], are out of the scope of this research. In those techniques, volume data is visualized by first creating a geometric representation such as isosurfaces in the volume and then displaying the surface.

Direct volume rendering algorithms can be classified in two main groups as *object-space* approaches and *image-space* approaches. In object-space approaches [82, 92, 98, 103, 93], the volume is processed in object-order, i.e., each volume element is processed in

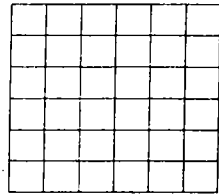
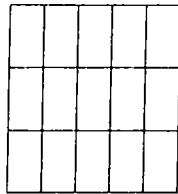
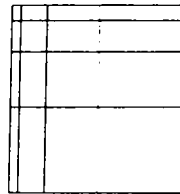
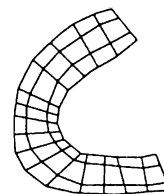
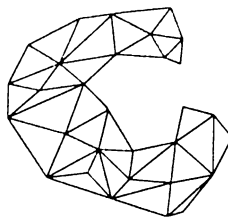
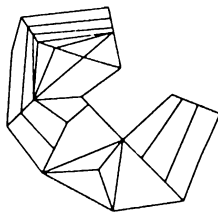
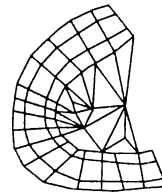
Structured Grids**Cartesian****Regular****Rectilinear****Curvilinear****Unstructured Grids****Regular****Irregular****Hybrid**

Figure 5.2: Types of grids encountered in volume rendering.

some order and its contribution to the pixels on the screen is calculated. In image-space approaches [54, 55, 92, 49, 29, 31, 88], the volume is processed in pixel-order, i.e., each pixel on the screen is processed in some order and contributions of the volume elements to this pixel location are calculated.

5.2 Ray-casting Based Direct Volume Rendering

Ray-casting based direct volume rendering (ray-casting DVR) [54, 55, 92] is an image-space approach in which a ray is cast from each pixel location and is traversed throughout the volume. An example of ray-casting DVR is illustrated in Fig. 5.3. The color value of the pixel is calculated by finding contributions of the cells intersected by the ray and by integrating these contributions along the ray. The traversal of ray through the volume and calculating the color of the pixel introduces two problems referred to here as *point location* and *view sort* problems. Efficient solution of these problems is crucial to the performance of the underlying algorithm. In the following sections, point location and view sort problems are described and existing approaches to resolve these problems are presented.

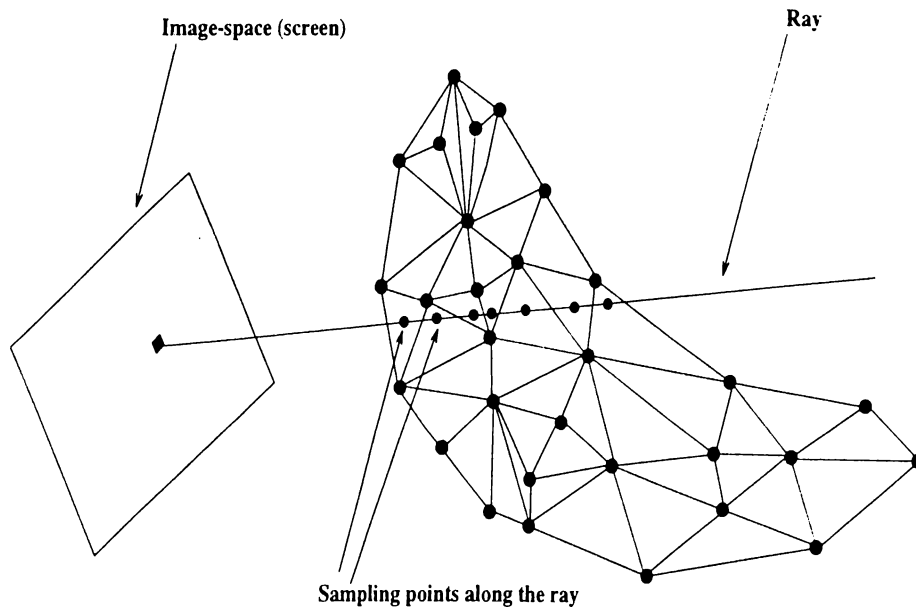


Figure 5.3: Ray-casting based direct volume rendering.

5.2.1 Point Location and View Sort Problems

Ray-casting DVR algorithms can be divided into two phases; *re-sampling* phase and *composition* phase.

In *re-sampling* phase, the ray is traversed through the volume and new sample points are taken along the ray. In unstructured grids, the ray is traversed in the volume to determine the list of cell intersections by the ray. Each ray-cell intersection means an entry point and an exit point of the ray through the cell. The entry and exit points are utilized to find the contribution of numerical data values at the vertices of the cell to the sample point(s) on the ray [29, 31, 49, 90]. In [90], for each ray-cell intersection, a new sample is computed at the midpoint of the ray between its entry and exit points on the cell (Fig. 5.4). The numerical data at vertices of the cell is interpolated to find the value at the new sample point on the ray. *Inverse distance interpolation* [90] can be used to calculate color and opacity values. First, distance of each vertex to the sample point on the ray is calculated. Then, contribution of each vertex to the new sample point is calculated inversely to the distance of the vertex to new sample point. That is, the smaller is the distance, the larger is the contribution of the vertex.

In *composition* phase, the contributions of the cells intersected by the ray are composited in a predetermined way either from *back-to-front* [54, 23], starting from the last cell intersected by the ray in 3-dimensional space, or from *front-to-back* [55, 80, 92], starting from the first cell intersected by the ray in 3-dimensional space. First, the scalar value(s) at each sample point on the ray is mapped to a color (C_s) and an opacity value (O_s) via applying a *mapping function*, also called *transfer function*, which converts numerical value to color and opacity to represent the characteristics of the physical environment and simulation results. The determination of right mapping function is out of the scope of this research. The color and opacity values are composited to form the color at the pixel on the screen. If composition is performed from back-to-front, following equation [54, 23, 96] is evaluated to find the composited color at the pixel

$$C_{i+1} = C_i(1 - O_s) + C_s O_s \quad (5.1)$$

where C_{i+1} is color after the sampling point on the ray, C_i is the color composited from previous sampling points on the ray, C_s is the color at the current sampling point on the ray and O_s is the opacity on at the current sampling point on the ray. Initially, a background with opacity $O_s = O_b = 1$ is placed behind the volume and $C_s = C_b = \text{background color}$ is taken as the starting point. If front-to-back composition is used, following equation [55] is evaluated to calculate the composited color

$$\begin{aligned} C_{i+1} O_{i+1} &= C_i O_i + C_s O_s (1 - O_i) \\ O_{i+1} &= O_i + O_s (1 - O_i) \end{aligned} \quad (5.2)$$

where C_{i+1}/O_{i+1} is the color/opacity after processing sample point, C_i/O_i is the color/opacity before processing sample point, and C_s/O_s is the color/opacity at the sample point. Initially, C_0 and O_0 are set to zero. Note that equations (5.1) and (5.2) are associative, but *not* commutative. Hence, the composition of the sample points should be done in a predetermined order. This restriction requires that either sample points on the ray should be sorted or ray-cell intersections should be determined in a sorted way.

Determining the volume element that contains the sample point on the ray in the re-sampling phase is called *point location* problem. For unstructured grids, it involves

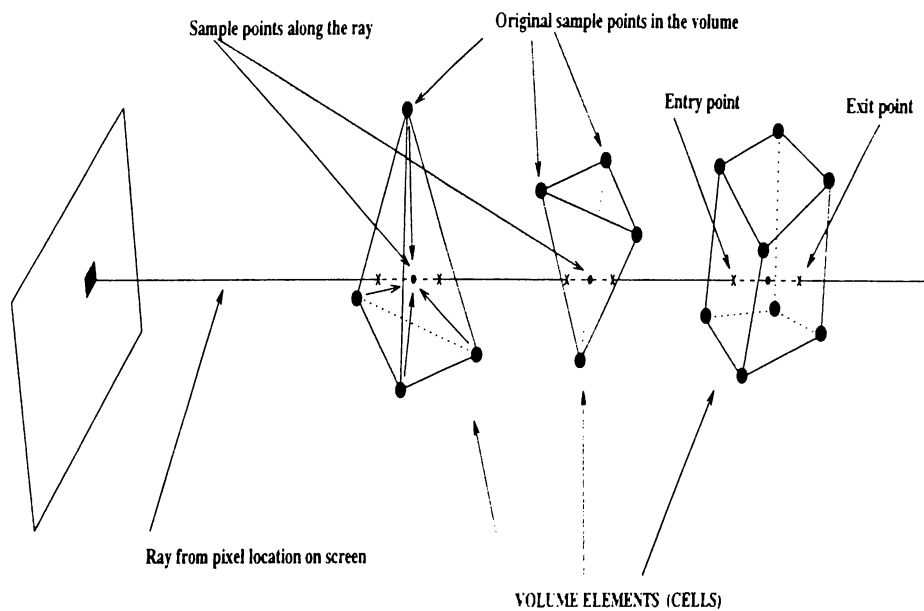


Figure 5.4: Re-sampling phase of the ray-casting DVR. The color and opacity values at the sample point on the ray are calculated by finding the contributions of original sample points which form the cell. After re-sampling, sample points on the ray are composited to generate the color on the screen.

finding the intersection of the ray with the cell. Sorting sample points on the ray or finding the intersections in a sorted order is defined as *view sort* problem. These problems are relatively easy to solve in structured *cartesian*, *regular*, and *rectilinear* grids. The regular distribution of data points over 3-dimensional space and implicit connectivity between volume elements make these problems almost trivial ones to solve. However, solving point location and view sort problems is more difficult in *curvilinear* and *unstructured* grids. In unstructured grids, data points (original sample points), hence volume elements, are distributed irregularly over 3-dimensional space. A naive algorithm may need to search all cells to find an intersection, thus requiring very large execution times for large data sets. In addition, sorting sample points on a ray takes a lot of time, if not handled efficiently, because many cells may be intersected by the ray. Therefore, the performance of the underlying algorithm closely depends on how efficiently it resolves these problems.

5.2.2 Approaches to Solve Point Location and View Sort Problems

In this section, existing work on point location and view sort problems on unstructured grids are summarized. These algorithms use the coherency in the volume and on the image-space to increase the efficiency of the algorithms.

Garritty [29] proposes an algorithm that utilizes connectivity between volume elements. First, rays are intersected with exposed (exterior) faces to find the first intersection of the ray with the volume. In order to further decrease the search for the first intersection, all exposed faces are geometrically sorted into a coarse 3-dimensional grid. The ray is intersected with this grid first and only the faces in the grid locations, which are intersected by the ray, are tested for intersection with ray. Once the intersection of the ray with a cell is found, this intersection is the entry point of the ray into the cell. Entry point of the ray to the next cell, which is the exit point of the ray from the current cell, is found by only considering the faces of the current cell. After an exit point is calculated, through the connectivity information between cells, next cell that shares the face, is also found. Since unstructured grids may be curved or may have voids in the

volume. if a ray exists the volume at a point, exterior faces are intersected with the ray again to find the next entrance point of the ray to the volume.

Koyamada [49] describes an algorithm for unstructured grids with tetrahedral volume elements. The ray-cell intersections are carried out using connectivity between volume elements. The first intersection of the ray with the volume is found by projecting and scan-converting the exterior face on the screen. In order to reduce the number of exterior faces scan-converted, only front-facing (for front-to-back traversal) or back-facing (for back-to-front traversal) exterior faces are projected and scan-converted. When scan-converting an exterior face, a ray is cast from each pixel generated and it is traversed in the volume. The next cell that the ray intersects is found by only checking the faces of the current cell. If the ray exists the volume, composited color and opacity values are stored to the pixel location. Note that when another exterior face is projected to the same pixel location, the composition step uses the color and opacities at the pixel location as initial values. For non-convex volumes, front-facing or back-facing exterior faces are assigned a depth-priority so that rays intersect these faces in order of increasing distance to the screen. The depth-priority is calculated by simply sorting the distance of centroids of the front-facing or back-facing exterior faces. Since this sorting is an approximate sorting, it may result in artifacts in the image due to incorrect priority. It is stated in the paper that these cases are rather rare and if needed more complex sorting algorithms such as list-priority algorithms [77] can be used.

Tabatabai *et al.* [88] describe methods to visualize volumes composed of *non-linear* elements. The intersection of the ray with a cell face is calculated by solving a set of linear equations. The first intersection of the ray is found by only considering exterior faces of the volume. The algorithm divides the image space into contiguous pixel regions of the same size. The projected bounding box of the face is used to mark regions as *possibly* containing this face. Hence, the ray-face intersection is calculated by only considering faces in a region. The next volume element intersected are found by using the connectivity between volume elements.

All of the above approaches require connectivity between volume elements to traverse the ray through the volume efficiently. If such connectivity is not given, the algorithm should construct such a relation as a preprocessing step. If connectivity does not exist

between cells, then efficiency of these algorithms may degrade considerably. The following two approaches do not require connectivity, thus provide more general solutions to point location and view sort problems.

Giertsen [31] utilizes a scan-plane buffer to solve point location and view sort problems. The scan-plane buffer is a 2-dimensional array and stores information within a plane perpendicular to one scanline on the screen, i.e., the scan-plane extends in x and z directions perpendicular to screen while screen extends in x and y directions in 3-dimensional space. In this algorithm, the z -dimension is also discretized in a sense due to scan-plane buffer. The algorithm proceeds from one scanline on the screen to the next. At each scanline, the intersection of scan-plane with volume elements is calculated. The volume elements that are intersected by this scan-plane are *sliced* by finding edge intersections of faces of volume elements with scan-plane. The intersection calculations from one scanline to the next one are done by incremental calculations using a list of *active cells*, whose y -extend covers the current scanline. In order to update active cell list for the current scanline efficiently and decrease the slicing calculations, two methods are proposed in the paper. In the first method, all the cells are bucket-sorted into a y -bucket as in scanline z -buffer algorithm in polygon rendering according to their minimum y value. The active cell list is updated using this y -bucket. Another method to build active cell list and decrease slice calculations is proposed for volumes with large opaque regions. In this method, the cells are bucket-sorted into a z -bucket according to their minimum z value. Then, the z -bucket is traversed in increasing order of z so that cell slices closest to the screen is inserted into the scan-plane buffer. If the foremost point of the slice is opaque, the algorithm only processes segments before the opaque slice. After volume elements are sliced for the current scanline, each slice is divided into triangles and each triangle is further decomposed into line segments in z direction. A line segment is stored into the scan-plane location corresponding to the foremost end of the line segment. In this way, view sort operations is done efficiently. A run-length encoding which shows the expected location for the next segment is also stored at the same location. Run-length encoding avoids extensive searching during composition phase. Giertsen uses Sabella's method [80] for color and opacity calculations in the composition phase. The composition is carried out by processing the line segments in front-to-back order and linearly

interpolating the ray along them. Since scan-plane buffer discretizes the 3-dimensional volume in x and z directions. The quality of images and performance of the algorithm depends on the discretization level (i.e., resolution of the scan-plane buffer).

Challinger [9, 10, 11] employs a scanline z-buffer based algorithm to solve point location and view sort problems. In the former work [9], a cell-by-cell approach is used. Intersection of the ray with cells is found using scanline algorithm. First, cells are bucket-sorted into a y-bucket as in scanline z-buffer algorithm. Then, algorithm processes each scanline starting from the lowest scanline on the screen. An active cell list is created for the current scanline using the y-bucket list. The active cells for the current scanline are bucket-sorted into an x-bucket with respect to their minimum x coordinate. When processing pixels on the current scanline, an active cell list is created for the current pixel using the x-bucket. In this way, the number of cells to be tested for intersection is reduced largely. In the latter works [10, 11], face-by-face approach is used. Cells are divided into faces and algorithm operates on the faces of the cells to find ray-face intersections. The algorithm is based on conventional scanline z-buffer hidden surface algorithm used in polygon rendering. The algorithm needs cells with planar polygonal faces as in the former work. Non-planar faces are broken into two triangular polygons. In this algorithm, instead of casting rays from pixels and finding their intersection with polygons, which make up the face of a cell, projection of polygons are processed (rasterized as in polygon rendering algorithms) in scanline-order to find ray-face intersections. Note that if a pixel in the current scanline is covered by the projection of a face, then the ray shot from that pixel intersects the corresponding face. In the first step of the algorithm, all polygons are bucket-sorted into a y-bucket according to their minimum y coordinate. As in the former work, the algorithm proceeds from one scanline to other scanline on the screen and from one pixel to the next in the same scanline. An active list of polygons are created for the current scanline using the y-bucket. The active polygons, whose y-extend covers the current scanline, are intersected with the current scanline to find edge intersections. The spans created by edge intersections are bucket-sorted into an x-bucket. As pixels are processed in the current scanline, an active edge list (or span list) is created using the x-bucket. The spans are rasterized to generate ray-polygon intersections at the current pixel. The distance of each ray-polygon intersection

and related information (e.g., a pointer to the cell) are inserted into a sorted linked list, called *intersection list* (referred to here as z-list), which is sorted in increasing distance values. This sorted list is utilized in composition phase. Note that two consecutive ray-polygon intersections in the z-list corresponds to entry and exit points of ray with the cell. During the composition phase for the current pixel, the list is traversed in order and each pair of intersections is used to find corresponding sample point on the ray. During traversal of the list, these sample points are composited. The algorithm uses image-space coherency to efficiently find the intersection of scanlines with cell faces and to avoid sorting the z-list for each pixel in the current scanline. Projections of the cell faces cover consecutive scanlines on the screen. Hence, the intersection of scanline with polygon edges can be carried out using incremental calculations. Each span in the current scanline covers consecutive pixel locations. Therefore, sorting of z-intersections with polygons are avoided as long as the list of polygons intersected by the ray does not change.

In addition to above algorithms used in ray-casting based DVR approaches, algorithms developed in *object-space* approaches can also be utilized to solve point location and view sort problems. In object-space approaches, the volume elements are view sorted instead of sorting ray-cell intersections. This sorted order of cells can be utilized to find ray-cell intersections efficiently. Williams [104] describes a method to view sort cells in linear time in number of cells and their faces. His algorithm constructs a directed acyclic graph using plane equations of faces, view point location and interconnectivity relation between cells. Then, a topological sort operation is carried out using either a *depth-first* or a *breadth-first* search. This sort produces the view sorted order of volume elements. He gives complete algorithm for convex meshes. Algorithm for non-convex meshes has few limitations and may not handle all non-convex meshes. The algorithm may result in cycles in the graph for non-convex meshes even if such a cycle does not exist.

5.3 Previous Works on Parallel Direct Volume Rendering of Unstructured Grids

This section summarizes the previous work on parallel direct volume rendering of unstructured grids. Most of the previous work on parallel direct volume rendering has been carried out for parallel direct volume rendering of structured grids. Some of these approaches and related references can be found in [58, 63, 87, 107, 59, 2, 52]. Utilization of parallel processing in direct volume rendering of unstructured grids has been investigated by few researchers [9, 10, 11, 57, 105, 60]. Although the previous works by Challinger [9, 10, 11] also address parallel implementations for curvilinear grids, the algorithms do not exploit the nature of curvilinear grids and are designed to handle unstructured grids as well.

Challinger [9, 10, 11] presents algorithms for BBN TC2000 multicomputer. The BBN TC2000 provides a distributed shared memory in the form of memory boards associated with each processor. Processors access to the shared memory locations through a network called Butterfly switch. The software library provides a task generation mechanism that generates the next task to be assigned to processors dynamically [99, 11]. In the former work [9], two algorithms are presented. The y-bucket and volume data is stored in a scattered fashion in the globally shared memory across the “local” memory blocks assigned to each processor. Each entry of the y-bucket corresponds to a scanline on the screen. The y-bucket is initialized by processing volume cells and inserting pointers at the bucket locations corresponding to lowest numbered scanline intersecting the cell. In the “single-phase” algorithm, each scanline on the screen is considered as a task. Dynamic task allocation on demand-driven basis is performed to assign scanlines to processors. In this scheme, each processor gets a scanline to render when it becomes idle. After receiving a scanline, each processor creates local x-buckets using the cells active at the current scanline. The local x-buckets, which are duals of y-buckets, are stored in the local memory blocks associated with each processor. Each processor, then, creates an intersection list for each cell active at the current pixel using the local x-bucket. The intersection list is then processed to perform composition. In “two-phase” algorithm, the sampling and composition steps are separated as two phases. Scanlines on the screen

are scattered to processors in a round-robin fashion statically. In the sampling phase, processors sweep through scanlines assigned to them and create intersection lists for each pixel on each scanline assigned to them. These intersection lists are stored in the local memories. In composition step, each of these intersection lists are processed to perform composition of sample values for the corresponding pixel. In two-phase algorithm, since intersection lists are saved, when a new transfer function is used to generate colors, only composition phase is executed. The main disadvantage of scanline based task generation is the low scalability. The scalability of the algorithm is limited by the number of scanlines on the screen. In the latter works [10, 11], image space is divided into square tiles which are considered as tasks and are assigned to processors dynamically. Volume data is scattered across the memory blocks associated to processors as in [9]. Since the algorithm employed to solve point location and view sort problems operate on faces, volume elements are decomposed into faces and face groups (groups of faces) are created in both parallel implementations. As is stated in the paper [10], structured grids are naturally decomposed into face groups since they are represented as three dimensional arrays. In the paper, decomposition of unstructured grids are very briefly mentioned but no specific algorithm is given. The creation of face groups and decomposition of cells into faces is done in parallel by dynamically assigning cells to processors. Vertices of the faces are transformed with respect to viewing parameters by assigning each row of the grid to processors dynamically. Then, cell faces are sorted according to image tiles they fall into. This sort is done in two passes due to inefficiency in shared memory allocation routines. In the first pass, number of faces crossing each tile is calculated using bounding boxes of the faces. Face groups are assigned to processors dynamically and each processor increments local counters corresponding to image tiles. Local counters, which are not zero, are added to global counters in shared memory after all cell faces processed. Necessary space for image tile buckets are then allocated in the shared memory. Note that if there are already allocated buckets in the memory, they are reallocated if their size is less than the current count of faces in that bucket. In the second pass, face groups are assigned dynamically as in the first pass to generate pointers to faces. Each processor now generates local pointers to faces for image tile buckets. These local pointers are then copied to the shared bucket lists. The viewing transformation and sorting steps

are followed by the rendering phase. There are two implementations for the rendering phase as in [9]. In the first implementation, composition and sampling phases are done simultaneously, whereas in the second implementation, intersection lists found in the re-sampling phase are stored into the local memories of processors to use when only transfer functions change. In both of the implementations, image tiles are assigned to processors dynamically. Image tiles, hence tasks, are sorted according to the number of cells associated to them. Larger tasks are assigned first to achieve better load balancing.

Williams [105] present algorithms for parallel volume rendering on Silicon Graphics Power Series (SGIPS) machine. The target machine is a shared-memory multicomputer with computer graphics enhancement through the use of graphics processors. The processors in SGIPS does not contain local memories and access to shared memory is done over a bus. The serial algorithms for direct volume rendering are based on object-space methods (such as projection and splatting). The cells are view sorted for proper composition by the view sort technique developed by Williams [105, 104]. The sorting technique, called meshed polyhedra visibility ordering (MPVO) algorithm, topologically sorts an acyclic directed graph generated from connectivity relation between cells. The topological sort is done by using either breadth-first search (BFS) or depth-first search (DFS) techniques on directed graph. Parallelization of the algorithms are divided into two stages: (1) the parallelization of generating directed graph used by the MPVO algorithm and (2) parallelization of topological view sort of the graph and rendering of the view sorted cells. Two algorithms are presented for stage (1). In the first algorithm, stage (1) is parallelized by assigning a cell to each processor to process. Each processor keeps local data structures (queues) to store the “source cell” used in the view sorting phase. These data structures are then merged and stored in the global memory. This scheme results in evaluating plane equation of shared faces twice. The second algorithm avoids this redundancy by evaluating the plane equation of the shared face only for the lower numbered cell of both cells sharing the face. This eliminates redundancy but introduces a search for each cell to find the desired faces and a separate sweep is required to update each cell accordingly. The parallelization of stage (2), i.e. view sort (based on BFS) and rendering (splatting based rendering) of cells, is done as follows for convex grids. In the first scheme, each processor takes a source cell from global queue and

splats it onto the screen. Since BFS on the graph produces cells that are spatially not overlapping, splatting of the cells can be done in parallel. Then, each processor finds the children of the source cell it splats and puts them into a local queue. When all source cells in the global queue is processed, local queues are merged into global queue. In the second scheme, two global queues are used. A single processor (`proc0`) performs BFS on the graph using source cells in the first global queue. This processor finds the children of all source cells in the first global queue and stores them in the second global queue, while other processors splat the cells in the first global queue. When all cells in the first global queue are processed and when `proc0` finishes constructing the first queue, pointers to global queues are exchanged. Thus, first global queue becomes second queue and second becomes first queue. If `proc0` finishes its work before others, it also helps splatting of the cells in the first queue. Parallelization of the algorithm for non-convex meshes is also presented. The MPVO algorithm for non-convex meshes requires DFS of the graph. One processor (`proc0`) performs DFS on the graph and the other processors perform the splatting of the cells. Two queues are used for this purpose. While `proc0` updates first queue, cells in the second queue are processed. Since cells need to be processed in the order they are output from the DFS routine, only limited amount of work can be parallelized such as transformation of cells and partitioning of cells for projection.

Lucas [57] describes a volume rendering algorithm for shared-memory multicomputers. The algorithm consists of two steps. In the first step, viewing transformations and lighting calculations are done. These calculations are performed on partitions of the volume data set. The data set is partitioned into rectangular regions. Unstructured data sets are partitioned by dividing the data recursively. Details of how to perform the subdivision is not given in the paper. The second step of the algorithm is the rendering of the volume partitions. In this step, screen is divided into non-overlapping rectangular regions and processors render one or more of the screen regions. Each screen region is processed in three steps; checking each partition if it falls into corresponding screen region, then checking each primitive in the partition for quick rejection of totally clipped primitives, and clipping and scan-converting primitives that overlap the partition. The effect of the number of screen partitions and number of volume partitions to the algorithm performance is examined to obtain an optimum division of the screen and volume

data set. It is unclear from the paper how screen regions are assigned to processors for achieving even load distribution.

Ma [60] present an object-space parallel algorithm for distributed memory multicomputers. This is the only known work on parallel volume rendering of unstructured grids on distributed-memory multicomputers. The multicomputer used in Ma's work is an Intel Paragon with 128 processors. In Ma's algorithm, the volume data is divided into P subvolumes, where P is the number of processors. The volume is considered as a graph and partitioned into subvolumes of equal number of volume cells (e.g., tetrahedrals) using Chaco graph partitioning tool [38]. The ray-casting volume rendering algorithm of Garrity [29] is used to render subvolumes in each processor. The subvolumes may have local exterior faces due to partitioning and it is possible that rays will exit from these faces and re-enter the volume from such faces, creating ray segments. The equations (5.1) and (5.2), used in the composition of colors and opacities, are associative, but *not* commutative. Thus, each processor inserts ray-segments (in sorted order) to linked lists. The partial images in each processor are composited to generate the final rendered image. In image-composition, screen is divided evenly into horizontal bands. Each processor is assigned a band to perform image-composition. The linked lists in each processor are packed and sent to respective processors for composition. Receiving processor unpacks the lists and sorts them. Then, these sorted lists are merged for the final image. Ma overlaps sending of ray segments with rendering computations to reduce the overhead of communication.

5.4 Discussion of Previous Works on Parallel Volume Rendering of Unstructured Grids

Most of the previous work on parallel rendering of unstructured grids evolved on shared-memory multicomputers [9, 10, 105, 57]. The algorithms developed in [105] can be considered as fine-grain algorithms and exploit the use of shared memory in the system. Load balancing is done dynamically assigning a cell to the idle processor for rendering. Such an assignment scheme will introduce a lot of communication overhead due to fine granularity of the assignments. In addition, parallel algorithms developed for sorting

the cells require a global knowledge of the database. Therefore, these algorithms are not very suitable for distributed-memory multicomputers.

The only work on distributed memory multicomputers is by Ma [60]. Ma uses object-space parallelism. The volume is partitioned using a graph partitioning tool into sub-volumes of equal number of elements. Unfortunately, the sequential rendering algorithm employed in the implementations is very slow. Thus, it hides many overheads of the parallel implementation. For example, image-composition operations take seconds even on large number of processors. In addition, composition time does not decrease linearly with increasing number of processors. This is basically due to sorting required on ray-segments for correct composition of colors and opacities. Moreover, even when viewing direction is fixed (to visualize volume under different transfer functions), inter-processor communication is still needed for image-composition.

Image-space parallelism is explored in [9, 10, 57]. Screen is subdivided into equal subregions and load balancing is achieved by dynamic allocation of subregions to processors [9, 10] or by scattered distribution [9]. The non-adaptive image-subdivision schemes in volume rendering has the same disadvantages as in parallel polygon rendering counterparts. Therefore, adaptive subdivision of the screen is a good alternative to non-adaptive subdivision.

Chapter 6

Spatial Subdivision for Volume Rendering

In this dissertation, we investigate image-space parallelism (section 3.2.1) for direct volume rendering of unstructured grids on distributed memory multicomputers. In this chapter, we present several algorithms for adaptive subdivision of the screen for efficient parallel volume rendering. Adaptive subdivision of the screen was only investigated in parallel polygon rendering algorithms [76, 99, 65, 26] and in ray tracing/casting [42, 5]. In volume rendering, screen is divided into equal subregions, which are assigned to processors either dynamically [9, 10] or by scattered distribution [9].

The common characteristic of the algorithms presented in this chapter is that they divide the image-space adaptively, using the primitives in the volume data, to achieve even distribution of rendering computations and data among the processors. The algorithms presented in this chapter can be grouped into two classes: 1-dimensional array based algorithms and 2-dimensional mesh based algorithms.

In the first group of algorithms, one-dimensional arrays for each dimension of the screen are used to represent the distribution of work load on the screen. The screen is partitioned using these arrays. Three algorithms are presented in this group: *Horizontal*, *rectangular*, and *recursive rectangular subdivision* algorithms. Horizontal subdivision algorithm is the simplest algorithm and divides the screen into horizontal bands of consecutive scanlines. Rectangular subdivision algorithm subdivides the screen rectangular regions. At the first stage, the screen is divided into horizontal bands. Then, each horizontal region is further divided vertically. Recursive rectangular subdivision algorithm is the most general type of these algorithms. It subdivides the screen recursively into

rectangular regions. This algorithm is based on the work by İşler [42, 5]. The implementation of the algorithm presented in this chapter slightly differs. During the subdivision, work load distribution arrays are updated using only the exchanged bounding boxes.

In the second group of algorithms, a 2-dimensional coarse mesh is superimposed on the screen. Subdivision algorithms use the work load distribution on this 2-dimensional mesh to subdivide the screen. Three algorithms are presented in this group: *Mesh-based adaptive hierarchical decomposition*, *Hilbert curve based subdivision*, and *graph partitioning based subdivision* algorithms. The first algorithm is based on Mueller's work [65]. This algorithm uses a summed area table [20] to subdivide the screen. The second algorithm is based on Hilbert space filling curve. In this algorithm, 2-dimensional mesh is traversed using the Hilbert space-filling curve [41]. This curve converts 2-dimensional representation into a one-dimensional array. This array is used to subdivide the screen. Hilbert curve and other space-filling curves have been used in various application areas [45, 83, 3, 69]. However, spatial-subdivision using Hilbert curve is a new approach in parallel rendering field. The last algorithm is a new algorithm and the most general type of these algorithms. In this algorithm, the subdivision of the screen is modeled as a graph partitioning approach. The 2-dimensional mesh is converted into a graph representation. This graph is partitioned using a state-of-the-art partitioning package, namely *Metis* [47].

After introducing the spatial subdivision algorithms, we experimentally compare and evaluate all algorithms with respect to *load balancing performance*, the *number of shared primitives*, and *execution time of algorithms*. In previous works on parallel polygon rendering [76, 99, 65, 26], the number of primitives in a region is used to represent the work load associated with that region. That is, screen is divided into regions and/or screen regions are assigned to processors using the primitive distribution on the screen. In all of these works, screen-space bounding box of a primitive is used to approximate the coverage of the primitive on the screen. This is done to avoid expensive computations to determine the exact coverage. In the experimental evaluation of the algorithms, the same approximations are used. That is, the number of primitives with bounding box approximation is taken to be the work load of a region for evaluating load balancing performance of the algorithms. The second criteria used in the comparisons is the number

of shared primitives after division of the screen. Shared primitives are the primitives that cross two or more regions assigned to different processors. Reducing the number of shared primitives is desirable since they potentially introduce overheads and waste system resources [42]. The most obvious is the waste of memory in the overall machine since such primitives have to be duplicated in different processors. They also introduce duplicated computations such as geometry processing in polygon rendering, intersection tests in ray tracing [42], etc. Execution time of the subdivision algorithms is another important criteria. A long execution time may take away all the advantages of a particular algorithm.

The algorithms proposed and presented in this chapter are also utilized for parallel implementation of a volume rendering algorithm. The sequential volume rendering algorithm is based on Challinger's work [9, 10]. This algorithm is a polygon rendering based algorithm. It requires volume elements composed of polygons and utilizes a scanline z-buffer approach to resolve point location and view sort problems. We discuss the application of the subdivision algorithms for this volume rendering algorithm. We present experimental speedup figures for rendering of some volume data sets on a Parsytec CC system installed in our department.

The rest of this chapter is organized as follows. Section 6.1 presents the spatial subdivision algorithms. Experimental comparison of these algorithms is given in section 6.2. The sequential volume rendering algorithm is presented in section 6.3. Parallelization of this sequential algorithm is presented in section 6.4. Section 6.5 presents the experimental results on Parsytec CC system.

6.1 Spatial Subdivision Algorithms

This section presents the subdivision algorithms covered in this work. In the following discussions, we assume that the number of primitives (based on the bounding box approximation) in a region is the work load of that region. The algorithms discussed in this section have three basic steps:

Step 1: Create screen space bounding boxes of the primitives. Initially, each processor

receives M/P primitives. Here, M is the total number of primitives and P is the number of processors. After receiving the primitives, each processor creates screen space bounding boxes of the local primitives. The bounding box of a primitive has the following structure:

```

structure bbox {
    short xmin, ymin;
    short xmax, ymax;
};

```

Here, $(xmin, ymin)$ is the lower left corner coordinates and $(xmax, ymax)$ is the upper right corner coordinates of the bounding box on the screen.

Step 2: Subdivide the screen into P regions using the primitive distribution on the screen. Each processor is assigned a single region after subdivision.

Step 3: Redistribute the local primitives according to screen subregions and processor-subregion assignments. In order to carry out redistribution step, each processor should know about the region assignments to other processors. For this reason, each processor receives screen subdivision information from other processors if such information is distributed among processors during subdivision.

After the redistribution of primitives, each processor renders the screen region assigned to that processor.

6.1.1 Horizontal Subdivision (HS)

In this scheme, the image plane is divided into P horizontal bands of consecutive scanlines. By allowing consecutive scanlines in each region, coherence in the image-space is preserved to some extent and the number of shared primitives is expected to decrease as there are less number of boundaries between processors. The division of the image plane is carried out using the distribution of work load in y-dimension of the image plane. An example of horizontal division is given in Fig. 6.1.

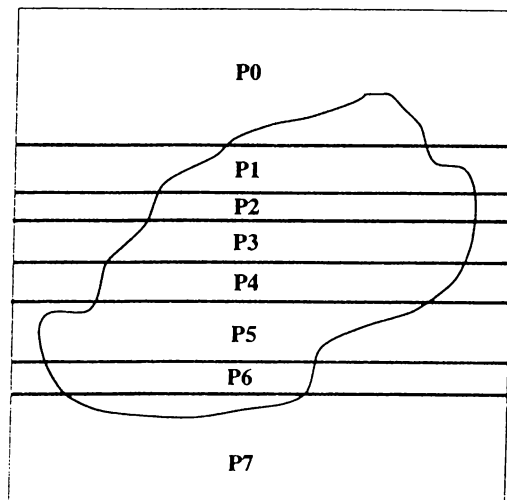


Figure 6.1: An example of horizontal subdivision for eight processors.

Basically, one-dimensional arrays are used to find the distribution of primitives over y-dimension of the screen. The first array is the *y-dimension local primitive start* (YPS_{local}) array of size N , where N is the resolution of the screen in y-dimension. The second array is the *y-dimension local primitive end* (YPE_{local}) array of size N . Each entry of these arrays corresponds a scanline on the screen. Each processor updates these arrays using local bounding boxes by the following algorithm.

```

for each local bounding box (bbox) k do
     $ymin = bbox[k].ymin; ymax = bbox[k].ymax;$ 
     $YPS_{local}[ymin] = YPS_{local}[ymin] + 1;$ 
     $YPE_{local}[ymax] = YPE_{local}[ymax] + 1;$ 
endfor

```

After all local bounding boxes are processed, $YPS_{local}[j]$ gives the number of local primitives that start at scanline j . Similarly, $YPE_{local}[j]$ gives the number of local primitives that end at scanline j . A global sum operation is performed on these two arrays so that each processor receives the global arrays YPS_{global} and YPE_{global} , containing the information for all primitives in the scene. Then, prefix sum operation is performed on each global array to obtain prefix sum arrays, YPS_{prefix} and YPE_{prefix} . The value $YPS_{prefix}[j]$ gives the number of primitives that start before scanline j , including the

scanline j . $YPE_{prefix}[j]$, on the other hand, gives the number of primitives that end before scanline j , including that scanline. Note that memory allocated for YPS_{local} and YPE_{local} can be reused for YPS_{prefix} and YPE_{prefix} arrays. The number of primitives in a region bounded by scanlines s and $e > s$ is given by the following equation:

$$\text{Number of primitives} = YPS_{prefix}[e] - YPE_{prefix}[s - 1]. \quad (6.1)$$

Note that this equation gives the exact number of primitives in a horizontal region bounded by $[s, e]$.

Processors subdivide the screen recursively using these prefix arrays until the number of regions is equal to the number of processors. In this way, a full binary tree, whose root being the whole screen, is conceptually generated. At each subdivision level i ($i = 1, \dots, \log_2 P$), a region bounded by scanlines s and e is divided into two regions $[s, j]$ and $[j + 1, e]$. The division line j that separates two subregions is determined, by checking all possible lines, such that the following expression is minimized.

$$\max(\text{workload}[s, j], \text{workload}[j + 1, e]) - \frac{M}{2^i}. \quad (6.2)$$

In this expression, function $\max(a, b)$ returns the maximum of a and b . The value $\text{workload}[s, j]$ gives the work load in the region bounded by scanlines s and j . In our case, work load is equal to the number of primitives in that region. Similarly, $\text{workload}[j + 1, e]$ gives the work load of the region bounded by scanlines $j + 1$ and e . The minus term $M/2^i$ represents the average load at subdivision level i . Here, M is the original number of primitives in the scene. Note that perfect load balance is achieved when each processor processes M/P primitives after redistribution step. In this respect, the minus term also represents the perfect load balance condition at each subdivision level. The expression given above also tries to decrease the number of shared primitives since the term $\max(\text{workload}[s, j], \text{workload}[j + 1, e])$ will be equal to $M/2^i$ when there are no shared primitives.

In the horizontal subdivision scheme, the atomic task is defined to be a scanline, i.e., scanlines are not divided. Due to this restriction, the scalability of horizontal division scheme is limited by the number of scanlines. In addition, the work load at each region is determined by the work load at each scanline. Hence, if there are large differences in

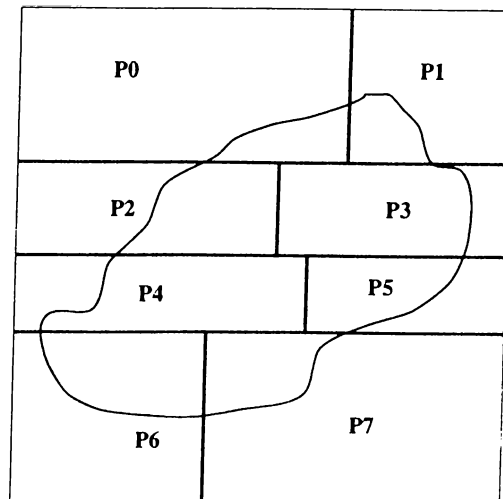


Figure 6.2: An example of rectangular division for 8 processors organized into 4 clusters and 2 processors in each cluster.

the work loads of scanlines, the load imbalance between regions may still be large. The limitations of the horizontal subdivision can be eliminated to some extent by using work load distribution in both dimensions of the screen. The scheme to implement this idea is given in the next section.

6.1.2 Rectangular Subdivision (RS)

In the rectangular subdivision scheme, processors are organized into a two dimensional $K \times L$ mesh, thus forming L clusters of K processors in each cluster. Then, the image plane is divided into L horizontal bands as in the horizontal subdivision. After partitioning image plane into L regions, the work load distribution in x-dimension in each region is calculated. Then, each region is divided into K vertical bands of consecutive vertical lines in x-dimension. An example of rectangular division scheme for 8 processors is illustrated in Fig. 6.2.

In this scheme, after L horizontal partitions are found, each processor treats each horizontal region as a new image plane rotated 90 degrees. Hence, the number of scanlines in each new image plane is equal to the number of vertical scanlines in x-dimension of the global image plane. Each processor uses the bounding boxes of local primitives to

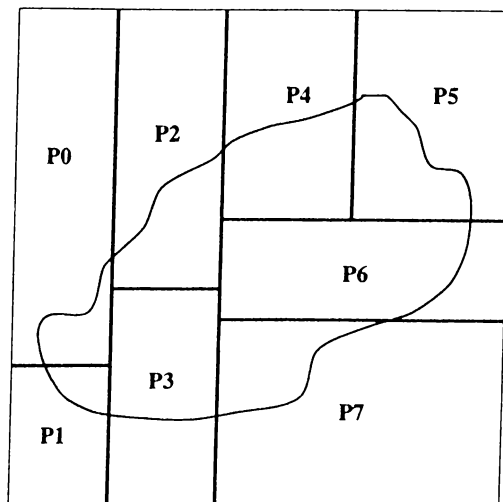


Figure 6.3: An example of recursive subdivision for eight processors.

find the work load distribution in each horizontal band. If a bounding box spans two or more horizontal regions it is divided into segments and work load distribution of each region is updated according to the corresponding segment. After this step, a global sum operation is performed to obtain the global work load distribution in x-dimension in each region. Afterwards, each processor finds vertical partitioning in the horizontal region of the cluster that processor belongs.

In order to redistribute primitives, processors need the vertical division information in other clusters so that they can find the rectangular region the bounding box of a local primitive crosses. At the last step, a global concatenate operation, on the vertical divisions in each cluster, is performed so that each processor has the information about vertical divisions in other clusters.

6.1.3 Recursive Rectangular Subdivision (RRS)

In this scheme, the image plane is divided into P rectangular regions in $\log_2 P$ steps. At each subdivision step i ($i = 1, \dots, \log_2 P$), the subregion assigned to a group of processors is divided into two new subregions either vertically or horizontally. An example of recursive subdivision is given in Fig. 6.3.

In this scheme, primitive distribution over two dimensions of the screen is required

to be able to divide the screen horizontally or vertically. This scheme uses the same data structures used in horizontal subdivision scheme. However, data structures used for horizontal load distribution are duplicated for vertical load distribution. That is, in addition to YPS_{local} and YPE_{local} arrays for y-dimension, each processor allocates XPS_{local} and XPE_{local} arrays for x-dimension of the screen. Initially, each processor is assigned the whole screen as its local image region. Each processor, then, updates its local copy of the YPS_{local} , XPS_{local} , YPE_{local} , and XPE_{local} arrays using the local bounding boxes as follows:

```

for each local bounding box (bbbox) k do
     $xmin = bbbox[k].xmin; xmax = bbbox[k].xmax;$ 
     $ymin = bbbox[k].ymin; ymax = bbbox[k].ymax;$ 
     $YPS_{local}[ymin] = YPS_{local}[ymin] + 1;$ 
     $YPE_{local}[ymax] = YPE_{local}[ymax] + 1;$ 
     $XPS_{local}[xmin] = XPS_{local}[xmin] + 1;$ 
     $XPE_{local}[xmax] = XPE_{local}[xmax] + 1;$ 
endfor

```

The work load distribution in two dimensions are obtained by performing global prefix sum operations on these arrays to obtain PS_{prefix} and PE_{prefix} arrays for each dimension of the screen. However, unlike the horizontal subdivision, the memory locations used for PS_{local} and PE_{local} arrays cannot be reused for prefix sum arrays in this scheme. The reason for this restriction will be apparent in the next paragraph. After performing global prefix sum, each processor divides its local image region into two subregions either horizontally or vertically. The division that achieves better load balance is chosen. Note that for the group of processors that are assigned the same image region, the division will be the same. After the division, half of the processors are assigned one of the regions, and the other half of the processors are assigned the other region. Following the region assignment, bounding boxes crossing the boundary between two regions are exchanged between neighbor processors assigned to the other subregion. Neighborhood between processors can be defined according to various criteria such as interconnection topology of the architecture, labeling of the processors etc. In this work, we chose hypercube labeling

for neighborhood definition since it is very simple. A processor k sends the local bounding boxes belonging to other region to the processor whose processor id is $k \oplus (2^{(i-1)})$ at subdivision step i . After this exchange operation, each processor has bounding boxes that projects onto its new local image region and the subdivision operation is repeated for new image region.

In order to subdivide the new region, we need to update PS_{local} and PE_{local} arrays for each dimension of the new subregion. We update these arrays incrementally using bounding boxes exchanged between processors. Each processor decrements the appropriate positions in PS_{local} and PE_{local} arrays for bounding boxes sent to the other processor and increments the appropriate locations in PS_{local} and PE_{local} arrays using received bounding boxes. In order to perform updates incrementally in this way, local PS_{local} and PE_{local} arrays should be maintained. Hence, prefix sum arrays cannot share the same memory locations with these arrays unlike horizontal subdivision scheme.

After $\log_2 P$ steps, each processor is assigned a unique rectangular region of the screen. A global concatenate operation is performed on these rectangular region information so that each processor receives the region assignments to be used in redistribution step. Note that recursive subdivision method is a superset of horizontal and rectangular subdivision schemes. Horizontal subdivision scheme is obtained if we avoid vertical divisions at all subdivision levels. Rectangular subdivision scheme is obtained if we avoid vertical divisions for $\log_2 L$ levels and perform only vertical divisions in the remaining $\log_2 K$ levels.

6.1.4 Mesh-based Adaptive Hierarchical Decomposition Scheme (MAHD)

In mesh-based adaptive hierarchical decomposition [65] scheme, bounding boxes of the local primitives are tallied to mesh cells after the mesh is superimposed on the screen. Some primitives may cross multiple cells. In order to decrease the errors due to counting such primitives many times, Mueller uses a simple heuristic. Each primitive increments the weights of each cell it intersects by a value inversely proportional to the number of cells the primitive crosses. In this heuristic, if we assume that there are no shared

primitives between screen regions, the sum of the weights of individual cells forming a region gives a value linearly proportional to the exact number of primitives in that region. However, shared primitives still cause some errors but it can be expected that such errors are less than counting such primitives multiple times while adding cell weights. Mueller also points out in the paper that this heuristic gives better results.

After all primitives in each processor are tallied, local mesh values are globally summed so that each processor receives the global mesh values representing the distribution of all primitives. Each processor then converts the global mesh into a summed area table (SAT) [20]. The summed area table can be generated by performing a prefix sum on each individual row of the mesh followed by a prefix sum on each individual column. The screen is subdivided into regions at cell boundaries recursively using this summed area table. The summed area table allows to find the work load in a rectangular region, whose corner points are (x_{min}, y_{min}) and (x_{max}, y_{max}) using the following expression:

$$\begin{aligned} load[(x_{min}, y_{min}), (x_{max}, y_{max})] = & SAT[x_{max}][y_{max}] \\ & - SAT[x_{max}][y_{min} - 1] \\ & - SAT[x_{min} - 1][y_{min}] \\ & + SAT[x_{min} - 1][y_{min} - 1]. \end{aligned} \quad (6.3)$$

At each subdivision step longer dimension of the intermediate region is divided. Dividing the longer dimension aims at reducing the length of the perimeter of the final regions as an attempt to reduce the number of shared primitives crossing the region boundaries. An example of MAHD is illustrated in figure 6.4.

In this scheme, resulting regions are rectangular and each region consists of adjacent cells.

6.1.5 Hilbert Curve Based Subdivision (HCS)

In this scheme, the 2-dimensional mesh is traversed in a predetermined way and locations of the mesh cells are mapped to a one-dimensional array. This array is used to subdivide the screen.

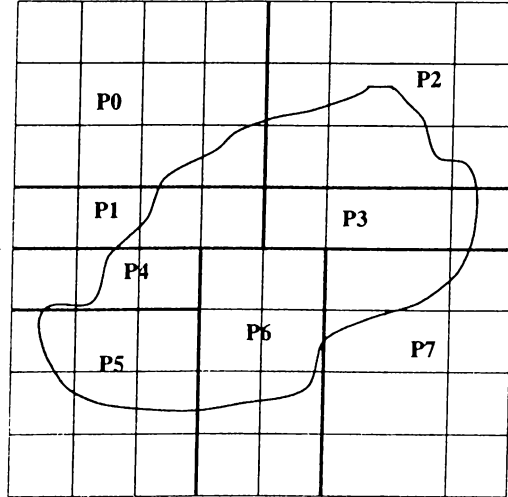


Figure 6.4: An example of mesh-based adaptive hierarchical decomposition for eight processors. Mesh resolution is 8×8 .

The curves, which are used to traverse the 2-dimensional mesh, belong to the set of *space filling curves* [69, 64]. Among various space filling curves [64], Hilbert curve is widely used in many applications. An example of traversing the 2-dimensional mesh with Hilbert curve and corresponding mapping of mesh cells to one-dimensional array are illustrated in Fig. 6.5. The numbers on each cell represents the indices of the array the cell is mapped. Mapping of two dimensional mesh indices to one dimensional array indices can be done by the algorithm given in [45]. The advantage of Hilbert curve over other space filling curves is that large jumps in the 2-dimensional mesh do not occur. This means that nearby cells are mapped to near locations on the 2-dimensional array. Therefore, we may expect that the length of the perimeter of the resulting regions will be less compared to the regions obtained by using other curves.

In HCS scheme, bounding boxes of primitives are tallied to mesh cells as in MAHD scheme. Like MAHD scheme, a bounding box contributes to a cell it intersects a value inversely proportional to the number of cells the bounding box crosses. Then, the mesh is traversed using hilbert space filling curve to map mesh locations to a one-dimensional array. A global prefix sum is performed on the one-dimensional arrays in each processor as in horizontal subdivision scheme. Afterwards, each processor divides this array into P regions such that each region has almost equal work load. The subdivision of the array

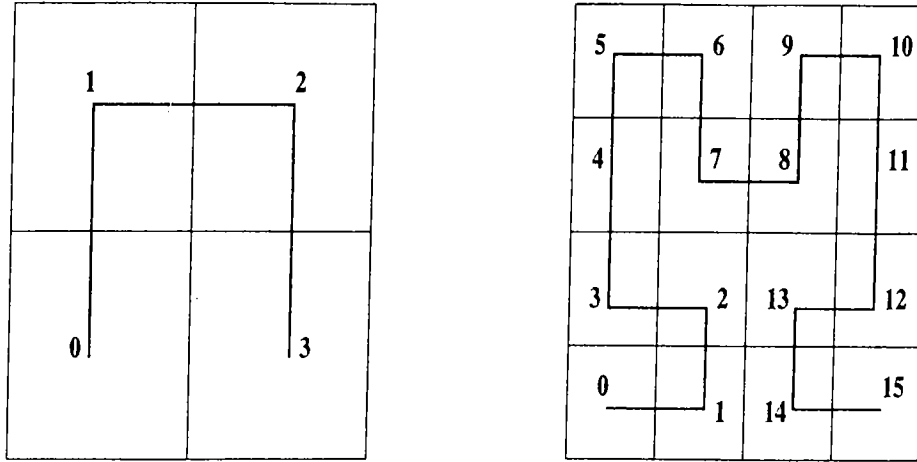


Figure 6.5: Traversing of the 2-dimensional mesh with Hilbert curve and mapping of the mesh cells locations into one-dimensional array indices.

corresponds to the subdivision of the image-space. An example of Hilbert curve based subdivision is given in figure 6.6.

In this scheme, resulting regions may be non-rectangular. However, they still consist of adjacent cells on the mesh.

6.1.6 Graph Partitioning Based Subdivision (GS)

This partitioning scheme models the spatial subdivision as a graph partitioning problem. Each cell in the mesh is assumed to be connected to its north, south, west and east neighbors. The vertices of the graph are the mesh cells and conceptual connections between mesh cells form the edges of the graph. The weight of a cell represents the number of primitives intersecting this cell. The weight of the edge between two cells represents the number of primitives crossing the boundary between these two cells. The objective in graph partitioning is to minimize the cut-size among the parts while maintaining the balance among the part sizes. Here, cut-size refers to the weighted summation of cut edges which connect more than one part. The size of a part refers to the weighted summation of the vertices in that part. In our case, balanced partitioning corresponds to maintaining computational load balance during rendering. Minimizing cut-size corresponds to minimizing the number of shared primitives. A state-of-the-art graph partitioning tool,

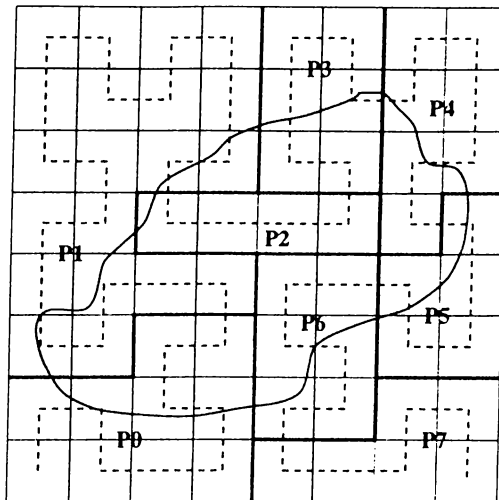


Figure 6.6: An example of Hilbert curve based subdivision for eight processors. Mesh resolution is 8×8 .

Metis [47], is used in this work. An example of graph partitioning based subdivision is given in figure 6.7.

In this subdivision scheme, each processor tallies local primitives and updates corresponding cell and edge weights. Cell weights are updated in the same way as in MAHD. Edge weight update scheme will be described in the next paragraphs. Each local graph is globally merged to obtain the graph representing the distribution of all primitives in the scene. After the global merge, each processor has the global graph to be partitioned. The mesh representation of the graph is converted into the representation used by Metis. In the original mesh representation, cell and edge weights are real numbers. These values are converted into integers since Metis operates on integer vertex and edge weights.

Metis uses multilevel partitioning approach consisting of three phases: coarsening, initial partitioning, and refinement. In the coarsening phase, the graph is coarsened down level-by-level to decrease the number of vertices by combining vertices to form new vertices. The coarsest graph is partitioned in the initial partitioning phase and this partitioning is refined in the refinement phase.

In the coarsening phase, various matching schemes can be used in Metis to combine appropriate vertices. The matching scheme used in this work is called *heavy edge matching*. In heavy edge matching scheme, at each level of coarsening, an unmatched vertex

formed by cells in neighboring rows or columns.

The graph partitioning approach subdivides the screen in the most general way. Unlike previous subdivision algorithms, noncontiguous sets of cells may be assigned to a processor. In addition, generated regions may be non-rectangular regions.

6.1.7 Redistributing the Primitives

After subdivision of the screen, each processor needs the primitives overlapping the region it is assigned in order to perform the rendering calculations. Thus, local primitives in each processors should be redistributed according to new regions and processor-region assignments.

Each processor classifies the local primitives according to the regions they overlap. According to the classification, each primitive is stored in the respective send buffer of that region. If a primitive overlaps multiple regions, the primitive is stored in the send buffers of those regions. These buffers are exchanged to complete redistribution of the primitives.

The subdivision algorithms HS, RS, RRS, and MAHD divide the screen into rectangular regions. The algorithm to classify the local primitives in these algorithms is given in figure 6.8. Since the regions are rectangular, the bounding box structure is used to represent regions for each processor. The variable *region*[*p*] denotes the region assigned to processor *p*.

The resulting regions in HCS and GS algorithms may be non-rectangular regions. Furthermore, regions may consist of disconnected mesh cells in GS algorithm. Therefore, the intersection test of the bounding box with the screen regions to classify the primitives will be more complicated for these algorithms. Instead, a different classification scheme is used in these subdivision algorithms. Just after the subdivision of the screen, each mesh cell is marked with the processor number whose screen region covers this particular cell. Note that each cell will be marked with a unique processor number. At the redistribution step, primitives are tallied to mesh cells as in subdivision step. During tallying of a primitive, the primitive is stored into the respective send buffers according to the marks of the cells the primitive covers. The algorithm to classify the primitives is given in figure 6.9.

```

for each bounding box (bbox) k do
  for each processor p do
    if (intersect(bbox[k], region[p]) == TRUE)
      Store the primitive k into the send buffer of processor p
    endfor
  endfor

intersect(bbox[k], region[p])
{
  if (bbox[k].xmin > region[p].xmax)
    return FALSE
  else if (bbox[k].xmax < region[p].xmin)
    return FALSE
  else if (bbox[k].ymin > region[p].ymax)
    return FALSE
  else if (bbox[k].ymax < region[p].ymin)
    return FALSE
  else
    return TRUE
}

```

Figure 6.8: The algorithm to classify the primitives at redistribution step of HS, RS, RRS, and MAHD algorithms.

The *stored* array (of size P) is used to prevent storing a primitive into the same send buffer multiple times. Initially, each entry of the array is set to -1 .

6.2 Experimental Comparison of Subdivision Algorithms

The algorithms presented in this chapter were implemented on a Parsytec's CC system using C language and PVM 3.3 [30, 72] for message passing. Each processing node of the CC system has 64 Mbytes of memory, each I/O node has 128 Mbytes of memory. Each node has PowerPC 604 processor running at 133 MHz. Experiments were done using two data sets called *blunt fin* and *post* data set. These data sets are used by many researchers in volume rendering field. Both blunt fin and post data sets are *curvilinear* sets. For the

```

for each bounding box (bbox) k do
  for each mesh cell c the bbox[k] covers do
    p = mark of the cell c
    if (stored[p] < k) then
      stored[p] = k
      Store the primitive k into the send buffer of processor p
    endif
  endfor
endfor

```

Figure 6.9: The algorithm to classify primitives in HCS and GS algorithms.

experiments and to be used in the rendering algorithm, which will be described in the next sections, these data sets were converted first into tetrahedrals [29, 82], by dividing each cell into five tetrahedrals, then into set of distinct triangles. Each triangle in the data set represents a face of a tetrahedral. The blunt fin data contains 381548 triangles and post data contains 1040588 triangles after conversion. All results presented in this section are the averages of results for two data sets obtained for three different viewing locations for each data set for the screen resolution of 512×512 . Figure 6.10 illustrates rendered images of data sets from one view used in the experiments.

We use the number of primitives in each processor to measure load balancing performance and to measure percent increase in the total number of primitives after subdivision. The load balancing values were calculated as $Max/Average$. Here, *Max* is the maximum of the number of primitives in each processor after subdivision. *Average* is the average number of primitives and is calculated by dividing the number of primitives in the scene before redistribution by the number of processors.

The subdivision algorithms MAHD, HCS, and GS use a 2-dimensional mesh superimposed on the screen for subdivision. The mesh resolution affects the performance of these algorithms. In addition, these algorithms calculate an estimated number of primitives in a region. This is because of the fact that a bounding box contributes to a cell a value inversely proportional to the number of cells the primitive crosses. This scheme will give exact number of primitives in a region only when there are no shared primitives in the region. The effects of these factors for these subdivision algorithms are

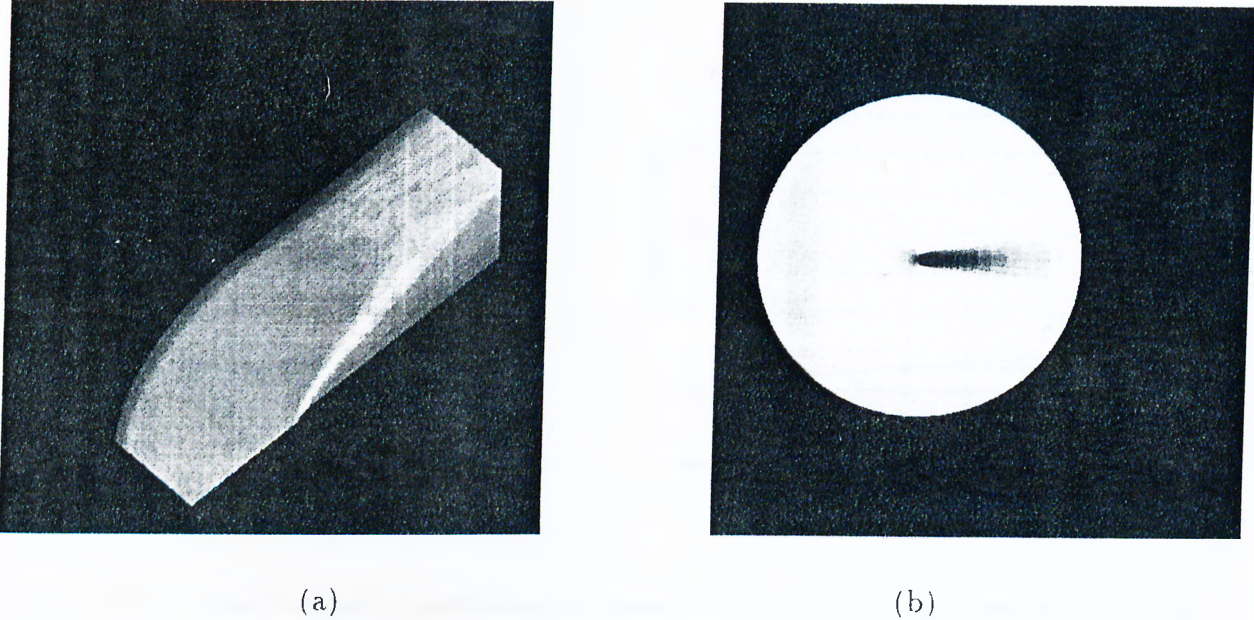


Figure 6.10: Rendered images of the data sets used in the experiments. (a) Blunt fin (381548 triangles, rendering time is 6.27 seconds on 16 processors) (b) Post data (1040588 triangles, rendering time is 8.55 seconds on 16 processors).

illustrated in figures 6.11 – 6.12. Figure 6.11 illustrates the load balancing performance of the algorithms, based on estimated loads in a region, for different mesh resolutions (of 32×32 , 64×64 , 128×128 , 256×256 , and 512×512 on 16 processors) and for different number of processors. We see that algorithms perform better as the resolution of the mesh increases. This is expected since higher resolution means more possible division lines and, thus, more search space. On different number of processors, HCS achieves almost perfect load balance. Note that MAHD subdivides the screen into rectangular regions while the screen is subdivided into non-rectangular regions in HCS scheme. As a result, HCS achieves better load balance than MAHD. We would expect GS algorithm to perform better than the other two since GS subdivides the screen in most general way. However, GS explicitly tries to reduce number of shared primitives using edge weights on the mesh graph. Minimizing number of shared primitives and achieving even work load distribution can be conflicting for some cases. MAHD implicitly tries to reduce shared primitives by dividing the longest dimension at each subdivision step, whereas HCS does not do anything to reduce number of shared primitives at all. This situation

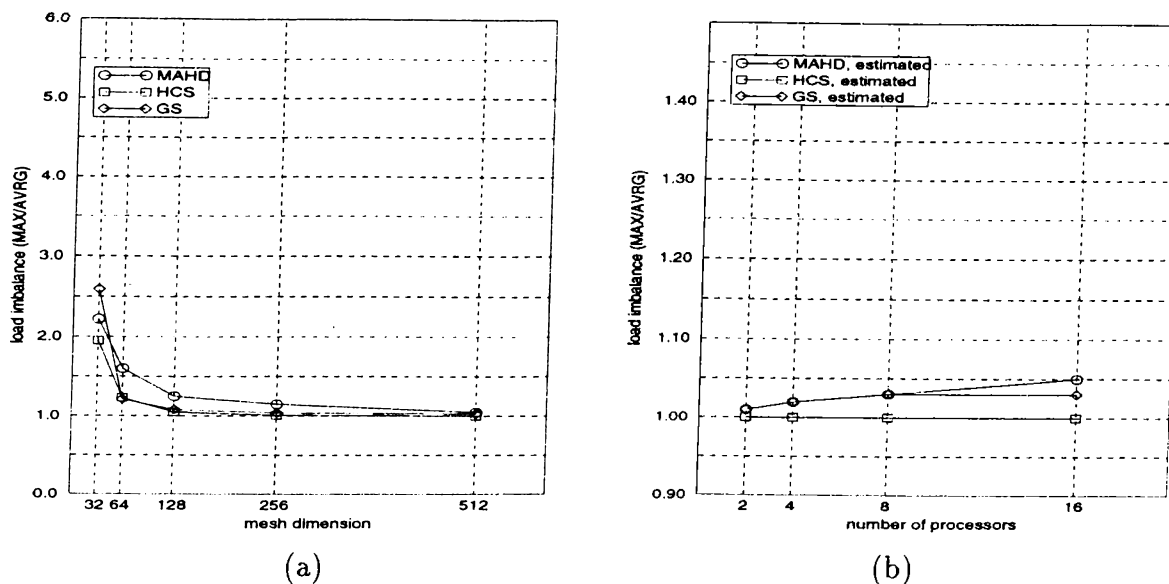


Figure 6.11: Load balancing performance, based on the approximate load calculations, of the MAHD, HCS, and GS algorithms (a) Different mesh resolutions on 16 processors. (b) Different number of processors.

is illustrated in figure 6.12. HCS gives highest percent increase in the number of primitives after redistribution while GS achieves lowest increase. In GS scheme, heavy edge matching was used as the matching algorithm, the number of vertices the graph should be coarsened down in the coarsening phase was taken as 100, the refinement algorithm used was Boundary Kernighan-Lin. These algorithms and values were chosen based on the observations in [47].

Figure 6.13 illustrates the load balance performance, based on the actual primitive distribution, of MAHD, HCS, and GS for different mesh resolutions and for different number of processors. MAHD achieves better load balance with increasing mesh resolution. GS and HCS, on the other hand, shows slightly different behavior. On 16 processors, both algorithms achieve best load distribution at the resolution of 128×128 . For different number of processors, GS achieves best load balance while the performance of HCS is the worst among the three algorithms. In addition, all algorithms achieve much worse load balance figures when compared with load balance figures based on estimated loads. As is seen in the figures 6.13(b) and 6.14, there exists a relation between number of shared primitives and the load balancing performance, based on the actual primitive

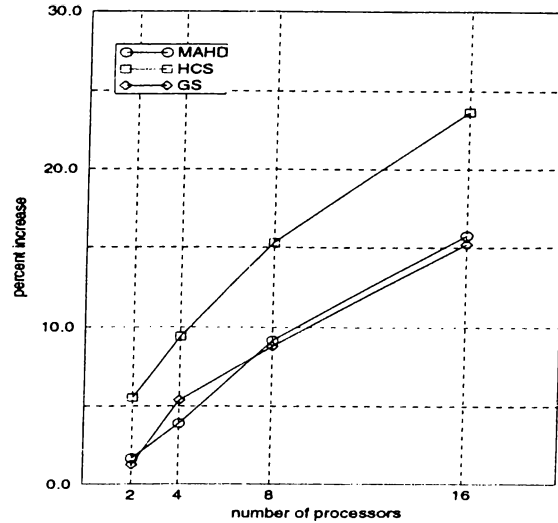


Figure 6.12: Percent increase in the number of primitives after primitive redistribution. Each value in the graph represents the percent increase in the total number of primitives for the mesh resolution the algorithm finds the best load balance, based on the estimated load distribution.

distribution, of the algorithms. Load balance graphs and graphs representing percent increase show a similar pattern for each subdivision algorithm. Shared primitives cause errors while calculating the number of primitives in a region. Hence, minimizing number of primitives tends to minimize errors incurred in load balancing calculations.

As is seen from figure 6.15, the execution time of the algorithms increases with mesh resolution as expected. The execution time in the figure is the sum of execution time for bounding box creation, division and redistribution phases. MAHD is the fastest among the three algorithms. MAHD involves simpler data structures and simpler operations for subdivision. In HCS, traversal of the mesh is required to map mesh cells into Hilbert curve. The GS scheme has an extra overhead of converting mesh graph to the graph representation of Metis. In addition, more volume of communication incurs in GS during combination of local meshes to form global mesh structure since mesh data structure includes edge weights between mesh cells in addition to mesh cell weights. For these reasons, execution times of GS and HCS are more sensitive to mesh resolution than that of MAHD.

Figure 6.16 illustrates the performance of all algorithms (HS, RS, RRS, MAHD, HCS, and GS) on different number of processors. We note that the load imbalance increases

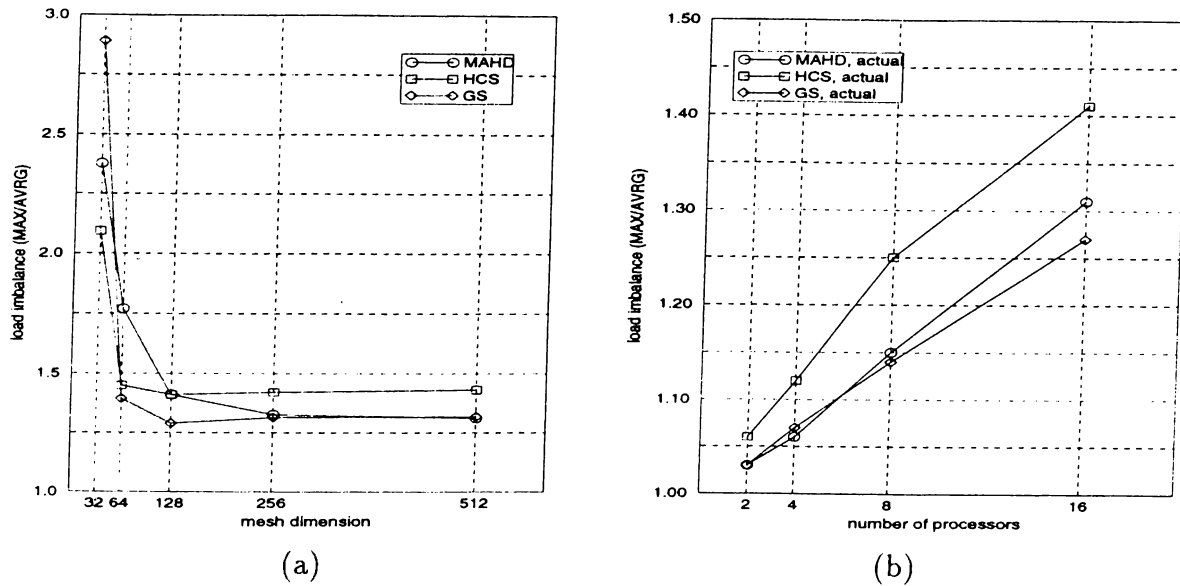


Figure 6.13: Load balancing performance, based on the actual primitive distribution, of the MAHD, HCS, and GS algorithms (a) Different mesh resolutions on 16 processors. (b) Different number of processors.

with increasing number of processors. This is expected since the subdivision is done on a finite resolution screen. As the number of regions increases (with increasing number of processors) resolution of each subregion decreases. Hence, subdivision algorithms have to operate on regions of lower resolution, severely restricting the search space. Worst load balance is achieved by HS and HCS algorithms. Although exact number of primitives in a region can be calculated in HS, allowing only horizontal subdivision lines restricts the search space of this algorithm. Among all of the algorithms, RRS achieves best load balance figures. Like HS and RS algorithms and unlike MAHD, HCS, and GS algorithms, the number of primitives in a region is calculated precisely in RRS algorithm. In addition, RRS divides the screen horizontally and vertically, whichever gives the better load balance at each subdivision step. This relaxes the restrictions on search space in HS and RS schemes to some extent. Figure 6.17 illustrates the percent increase in the number of primitives after redistribution for all algorithms on different number of processors. We observe a similar pattern to that of load balance figures in this case as well. The HS and HCS algorithms give the highest percent increase. For HS algorithm, this is basically due to the fact that the length of the perimeter of the subregions is

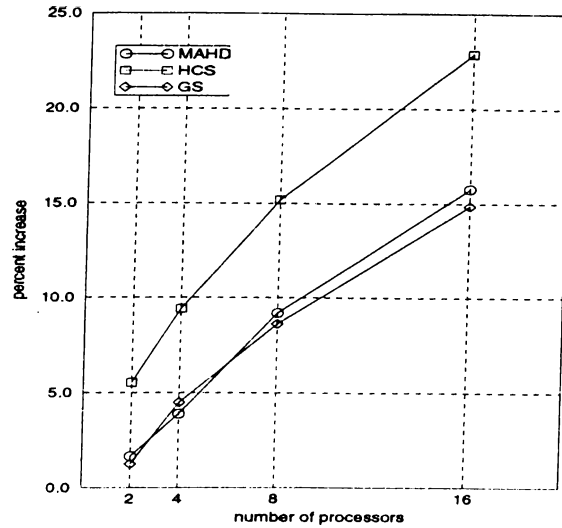


Figure 6.14: Percent increase in the number of primitives after primitive redistribution. Each value in the graph represents the percent increase in the total number of primitives for the mesh resolution the algorithm finds the best load balance, based on the actual distribution of primitives.

greater than that in RS and RRS algorithms.

Figure 6.18 illustrates the execution time of all algorithms on different number of processors. The execution time includes bounding box creation, division time, and primitive redistribution time for each algorithm. The values for MAHD, HCS, and GS represent the execution time of respective algorithm for the mesh resolution the algorithm achieves the best load balance. As is seen from the figure, HS is the fastest among all algorithms, since it is the simplest algorithm. We also observe that GS and HCS algorithms are faster than MAHD although they involve more overheads and more complex structures. This is due to the fact that MAHD achieves the best load balance on mesh resolution of 512×512 (except for 2 processors on which it achieved the best load balance on 256×256 mesh resolution), whereas GS and HCS algorithms achieve the best load balance at lower mesh resolutions. For example, GS achieves the best load distribution at mesh resolution of 128×128 for all number of processors. In general, we see that execution time of all algorithms decreases as the number of processors increases. However, starting from 8 processors the decrease in the execution time starts saturating. Table 6.1 illustrates dissection the execution time of each algorithm on different number of processors. As is seen from the table, the bounding box execution time decreases almost linearly with

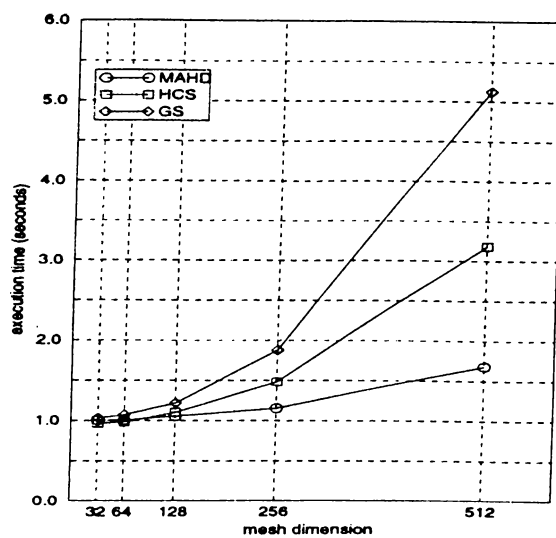


Figure 6.15: Execution time of MAHD, HCS, and GS for different mesh resolutions.

increasing number of processors. However, division time and redistribution times do not decrease as such. The communication overheads in division and redistribution steps do not decrease with increasing number of processors.

Table 6.1: Dissection of execution time of each algorithm on different number of processors. Timings are in seconds.

P	HS			RS			RRS		
	box	division	redist.	box	division	redist.	box	division	redist.
2	1.43	0.04	0.75	1.43	0.13	0.75	1.43	0.40	0.75
4	0.75	0.03	0.62	0.75	0.09	0.59	0.75	0.38	0.59
8	0.39	0.03	0.55	0.39	0.09	0.55	0.39	0.35	0.52
16	0.21	0.04	0.67	0.21	0.09	0.63	0.21	0.33	0.61

P	MAHD			HCS			GS		
	box	division	redist.	box	division	redist.	box	division	redist.
2	1.43	0.42	0.75	1.43	2.60	2.27	1.43	0.51	0.95
4	0.75	1.00	0.59	0.75	0.62	0.86	0.75	0.40	0.70
8	0.39	0.86	0.54	0.39	0.23	0.59	0.39	0.35	0.55
16	0.21	0.81	0.63	0.21	0.24	0.62	0.21	0.37	0.61

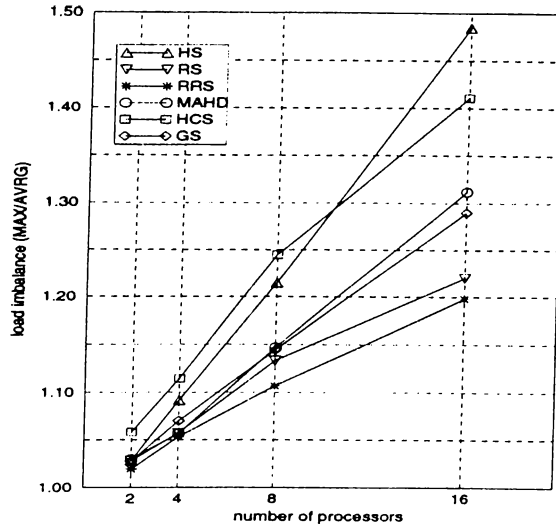


Figure 6.16: Load balance performance of all algorithms (HS, RS, RRS, MAHD, HCS, and GS) on different number of processors

6.3 Volume Rendering of Unstructured Grids: A Scanline Z-buffer Based Algorithm

The algorithm chosen for rendering is based on the algorithm developed by Challinger [10, 11]. This algorithm adopts the basic ideas in standard polygon rendering algorithms to perform hidden-surface removal operations efficiently. As a result, the algorithm requires that volumetric data set is composed of cells with planar faces. In this work, it is assumed that volumetric data set is composed of tetrahedral cells. If a data set contains volume elements that are not tetrahedral, these elements can be converted into tetrahedral cells by subdividing them [29, 82]. A tetrahedral has four points and each face of the tetrahedral is a triangle (Fig. 5.1 (b)), thus easily meeting the requirement of cells with planar faces. Since algorithm operates on the polygons, the tetrahedral data set is further converted into distinct triangles. Only triangle information is stored in the data files. The algorithm reads and performs rendering operations on triangles in this work.

The algorithm proceeds from one scanline to other scanline on the screen and from one pixel to the next in the same scanline. Basic steps of the algorithm is given below:

Step 1: Read volume data. In our case, the algorithm reads triangles representing faces

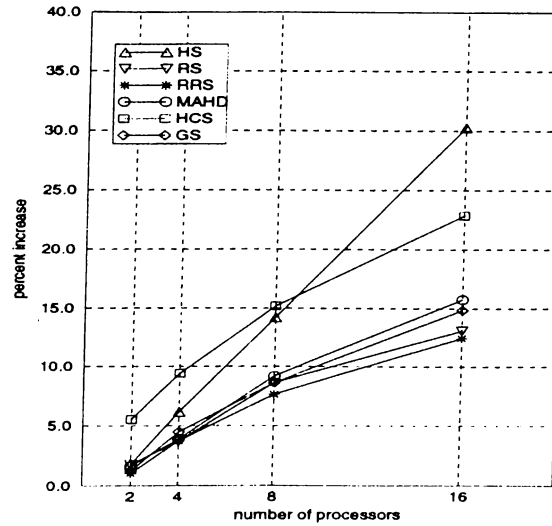


Figure 6.17: Percent increase in the number of primitives after redistribution for all algorithms on different number of processors. Each value in the graph represents the percent increase in the total number of primitives for the mesh resolution the algorithm finds the best load balance, based on the actual distribution of primitives.

of tetrahedrals from the data files.

Step 2: Transform the triangles into screen coordinates by multiplying each vertex by 4×4 transformation matrix. Perform *y-bucket sort* on the triangles. The *y-bucket* is one dimensional array of pointers that point to triangles of the input database. Each entry of the y-bucket corresponds to a scanline on the screen and a linked list of pointers is stored at each entry. The pointer to the triangle is inserted at the entry which corresponds to the lowest numbered scanline that intersects the triangle.

Step 3: Update *active polygon* and *active edge* lists for each new scanline, starting from the lowest scanline and continuing in increasing scanline number. The *active polygon* list stores the triangles that are starting and continuing at the current scanline. Before processing the current scanline, the corresponding entry of the y-bucket is inspected for new triangles. If there are new triangles, they are inserted into active polygon list. At the end of processing the scanline, triangles that end at the current scanline are deleted from the active polygon list. The *active edge* list stores the triangle edges that are intersected by the current scanline. Edges of triangle in the active polygon list are tested

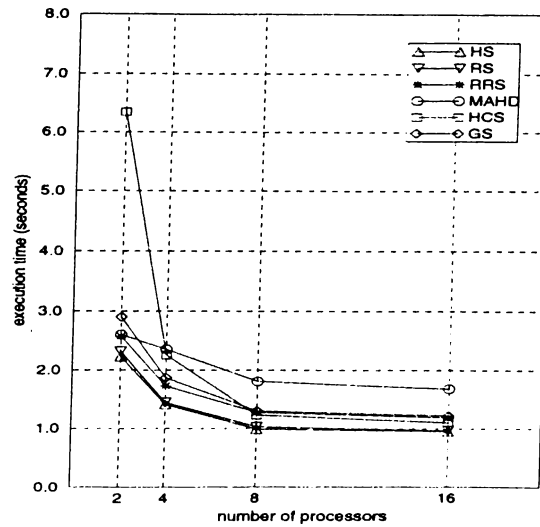


Figure 6.18: Execution time of all algorithms on different number of processors. For MAHD, HCS, and GS algorithms, the values represents the execution time of the respective algorithm for the mesh resolution the algorithm achieves the best load distribution.

for the intersection. Note that if a triangle is already in the active polygon list, then a pair of its edges is in the active edge list. For such triangles, new edge intersections are calculated incrementally using the edge information in the active edge list.

Step 4: For each active edge pair for the current scanline, generate a span, clip the span to the region boundaries, and put it in *x-bucket*. The *x-bucket* is one dimensional array of pointers. Each entry corresponds to a pixel location on the current scanline and stores a linked list of spans starting at that pixel location.

Step 5: Update *z-list* for each new pixel on the current scanline. The *z-list* is a linked list and each entry of the *z-list* stores the *z*-intersection of the triangle with the ray shot from the pixel location, span information, a pointer to the triangle, and a flag to indicate whether the triangle is an exterior or an interior face. Note that two consecutive triangles, if at least one of them is interior triangle, make up the corresponding tetrahedral cell in the volume. Hence, during the composition step, two consecutive triangles can be used for the determination of the sampling points on the ray. The *z*-intersections are calculated by processing the spans stored in the *x-bucket*. The *z*-intersections are updated incrementally by rasterizing spans. Each *z*-intersection is inserted into the *z-list*

in such a way that the list remains sorted in increasing z -intersection values. The z -list can also be considered as an active span list because span information for only spans that are active at the current pixel location is inserted into the list. Note that as long as no new spans are inserted, there is no need to sort list again for the next pixel.

Step 6: Composite the sample values for the current pixel location using z -list ordering. Repeat steps 3–6 until all scanlines and pixels are processed.

The algorithm uses image-space coherency for efficiency. The calculations of intersections of polygons with the scanline, insertion and deletion operations on the active polygon list are done incrementally. This type of coherency is referred to here as *inter-scanline* coherency. For each pixel on the current scanline, the intersection of the ray shot from the pixel and spans that cover that pixel are determined and put into the z -list, which is a sorted linked list, in the order of increasing z -intersection values. The z -intersection calculations, sorting of z -intersection values, insertion to and deletion from z -list are done incrementally. This type of coherency is referred to here as *intra-scanline* coherency.

6.4 The Parallel Algorithm

The parallel algorithm consists of the following basic steps:

Step 1: Read volume data. Initially, each processor receives M/P triangles. Here, M is the total number of triangles and P is the number of processors.

Step 2: (Subdivision phase) Create bounding boxes of the local triangles.

Step 3: (Subdivision phase) Divide the screen and redistribute the local triangles according to new screen regions and processor-region assignments.

Step 4: (Rendering phase) Perform steps 2–6 of the sequential algorithm on the local

screen region.

Steps 1–3 are the same steps of the subdivision algorithms. At step 4, the sequential algorithm steps are executed on local screen region using local primitives (triangles in our case) after primitive redistribution.

The screen is divided into regions using one of the spatial subdivision algorithms described in the previous sections. Determining the actual computational work load in a region is crucial to achieve even distribution of computational load among processors. As is stated in the previous sections, number of primitives are used to approximate the work load in a region in all previous works [76, 99, 65, 26]. We used the same approximations in the experimental comparison of the spatial subdivision algorithms. However, in the sequential and parallel algorithms given above, there are three parameters that affect the computational work load in a screen subregion. First one is the number of triangles, because the total work load due to insertion operations into y-bucket and insertions into and deletions from active polygon list are proportional to the number of triangles in a region. The second parameter is the number of scanlines each triangle extends. This parameter represents the computational work load associated with the construction of edge intersections – hence, corresponding spans –, clipping of spans to region boundaries, and insertion of the spans into x-bucket list. The total number of pixels generated by rasterization of these spans is the third parameter affecting the computational load in a region. Each pixel generated adds computations required for sorting, insertions to and deletions from z-list, interpolation and composition operations. The operations on each parameter takes different amount of time. Therefore, the work load (WL) in a region can be approximated using Eq. (6.4).

$$WL = aN_T + bN_S + cN_P \quad (6.4)$$

where N_T , N_S , and N_P represent the number of triangles, spans, and pixels, respectively, to be processed in a region. The values a , b , c represent the relative computational costs of operations associated with triangles, spans, and pixels, respectively. Finding exact number of pixels and spans generated in a region due to an overlapping primitive requires rasterization of the triangle. In order to avoid this overhead, the bounding box

approximation is used for pixels and spans. That is, a triangle with a bounding box with corner points (x_{min}, y_{min}) and (x_{max}, y_{max}) is assumed to generate $y_{max} - y_{min} + 1$ spans and $(y_{max} - y_{min} + 1) \times (x_{max} - x_{min} + 1)$ pixels.

The subdivision algorithms presented in the previous sections utilize only the number of triangles for work load. Incorporating the pixels and spans to these algorithms is accomplished as follows: Basically, each span and pixel generated due to bounding box of the triangle are treated as triangles with computational loads of b and c , respectively. That is, for a triangle whose bounding box has corner points (x_{min}, y_{min}) and (x_{max}, y_{max}) , there is one triangle with computational load of a , there are $y_{max} - y_{min} + 1$ triangles, whose height is one pixel and width is $x_{max} - x_{min} + 1$, each with computational load of b , and there are $(y_{max} - y_{min} + 1) \times (x_{max} - x_{min} + 1)$ triangles, whose height and width are one pixel, each with computational load of c .

6.5 Experimental Results

The parallel algorithm described in the previous section is implemented on Parsytec CC system using C language and PVM for message passing. This section presents experimental speedup results on two data sets *blunifin* and *post*, which are used in the experimental comparison of the spatial subdivision algorithms. All results presented in this section are the averages of results for two data sets obtained for three different viewing locations for each data set for 512 screen resolution.

Figure 6.19 illustrates speedup for only rendering phase (step 4 of the parallel algorithm) obtained when only the number of triangles are used to approximate work load in a region. In this case, maximum speedup obtained is 5.24 on 16 processors. Figure 6.20 illustrates speedup for the rendering phase when spans and pixels are incorporated into the subdivision algorithms. In this case, speedup increases to 11.44 on 16 processors, which is more than double the speedup when only the number of triangles is considered.

Figure 6.21 illustrates the speedup values when execution time (in seconds) of subdivision algorithms are included in the running times. We observe that speedup values

degrades in this case. This is expected since performing the subdivision introduces overhead to parallel execution. The maximum speedup achieved is 9.40 on 16 processors.

As is seen in figures 6.20 and 6.21, best speedup values are achieved by HS algorithm. This is an unexpected result since HS algorithm is the most restricted algorithm in terms of search space among other algorithms. However, this algorithm has advantage over the other algorithms for the volume rendering algorithm chosen in this work. It only disturbs *inter-scanline* coherency. It does not disturb *intra-scanline* coherency since screen is not divided vertically in HS scheme. Hence, HS incurs overheads due to inter-scanline coherency in step 4 of the parallel algorithm. However, other algorithms disturb both coherence incurring more overheads in step 4 of the parallel algorithm. In addition, bounding box approximation used for spans and pixels is likely to introduce more errors when screen is divided in horizontally and vertically than it is divided only horizontally. For example, the number of spans in a region can be calculated more precisely when only horizontal division lines are allowed. However, when vertical divisions are also allowed, bounding box approximation for the number of spans in a region introduces errors. Figure 6.22 illustrates such a case. As is seen in the figure, 9 spans are added to the work load of region **A** due to bounding box approximation while there are only 2 spans actually in that region.

The speedup values are not very close to linear. One of the reasons for this deviation from linear speedup is the bounding box approximation for triangles. The number of spans and pixels generated due to a triangle are calculated erroneously. Thus, the work load in a region calculated using bounding boxes does not truly reflect the actual work load. The second reason is that determining relative work loads of a triangle, a span and a pixel (i.e., constants a , b , and c in equation 6.4) is not easy. These values should be determined experimentally and the operations involving triangles, spans and pixels are not separated by solid boundaries. It is difficult to separate operations exactly related to a triangle, a span, and a pixel in the implementation. In our experiments, these values were determined after several trials using one viewing direction on *blunt fn* data, and the values that gave the best speedup on 16 processors were chosen. Another important reason for the deviation is the characteristics of data sets used in the experiments. In

these sets, many of the triangles are very small (even as small as one pixel) and the projection of triangles are clustered towards small regions of screen. Thus, subdivision algorithms should operate on small regions which restrict their search space.

Figure 6.23 illustrates the rendering times in seconds. Only rendering times (excluding overhead of subdivision) are given in figure 6.23(a). The running time of the sequential algorithm is 53.09 seconds. On 16 processors, rendering time drops to 4.78 seconds using HS algorithm. When subdivision overhead is included (figure 6.23(b)) parallel execution time on 16 processors increases to 5.66 seconds using HS algorithm.

6.6 Conclusions

In this work, image-space parallelism for volume rendering of unstructured grids has been investigated. Several adaptive subdivision algorithms were presented in this chapter. These algorithms can be classified into two groups: algorithm based on one-dimensional arrays and algorithms based on two-dimensional mesh. Horizontal subdivision (HS), rectangular subdivision (RS), and recursive rectangular subdivision (RRS) algorithms belong to first group, while mesh-based adaptive hierarchical decomposition (MAHD), Hilbert curve based subdivision (HCS), and graph partitioning based subdivision (GS) algorithms are the algorithms in the second group.

If the number of primitives in a region is taken as the work load of the region, our experimental results show that

- Among the algorithms in the second group, GS performs better than the other two algorithms. Since subdivision is modeled as graph partitioning in this scheme, It has larger search space than the other algorithms.
- There exists a relation between load balancing performance of MAHD, HCS, and GS and the number of shared primitives in a region. When number of shared primitives decrease algorithms achieve better load balance. These algorithms calculate the number of primitives in a region approximately. The shared primitives cause errors in these approximate amounts. Hence, they affect the load balance.
- Among all algorithms, RRS algorithm is the best in terms of load balance and

it results in lowest number of shared primitives. The better performance of this algorithm is due to the fact that the number of primitives in a region can be calculated exactly unlike MAHD, HCS, and GS. In addition, the algorithm divides the screen horizontally and vertically. Thus, it has larger search space than RS and HS algorithms.

These algorithms were employed in the parallelization of a volume rendering algorithm. It has been observed that only the number of primitives in a region does not provide a good approximation to actual computational load. The number of spans and pixels generated during the rendering of primitives were incorporated into the algorithms to approximate work load better. It has been experimentally shown that speedup values are almost doubled using these additional factors. On the average, we can render the data sets used in the experiments in about 6 seconds on 16 processors of Parsytec CC system.

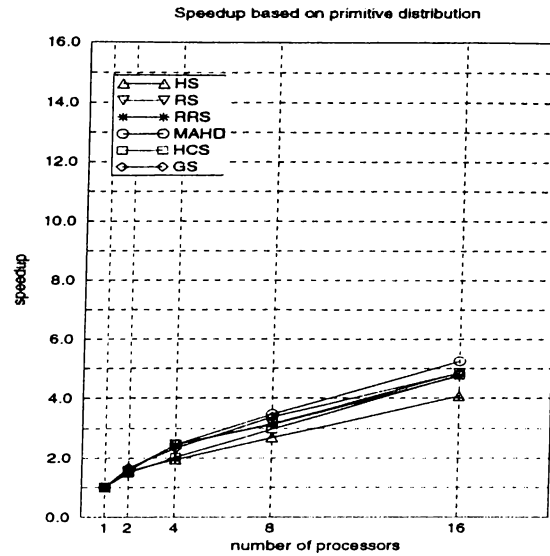


Figure 6.19: Speedup for only rendering phase when only the number of triangles in a region is used to approximate work load in a region.

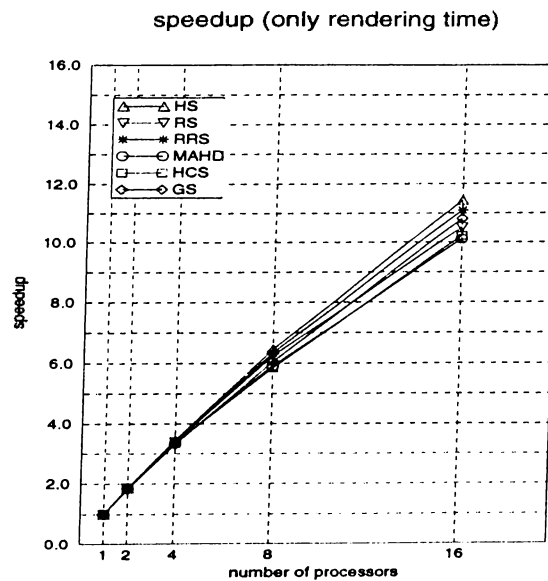


Figure 6.20: Speedup for rendering phase (step 4 of the parallel algorithm) when spans and pixels are incorporated into the subdivision algorithms.

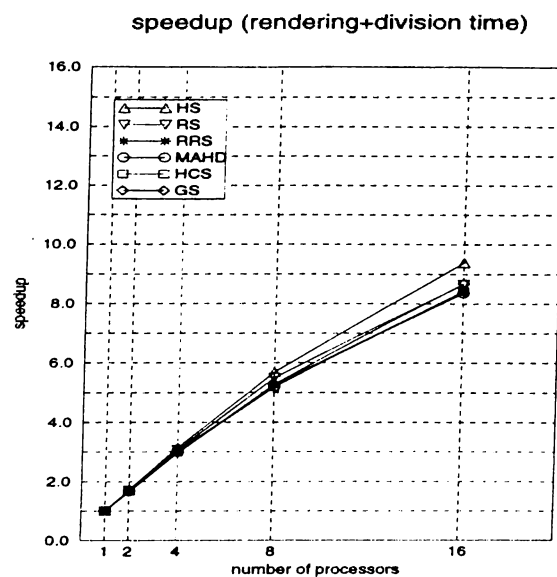


Figure 6.21: Speedup values including the execution time of subdivision algorithms.

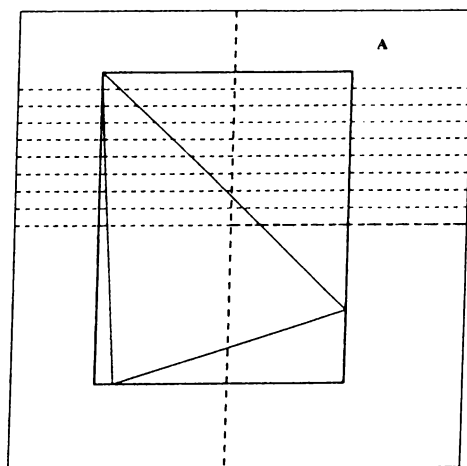


Figure 6.22: Errors due to bounding box approximation in calculating the number of spans when vertical divisions are allowed.

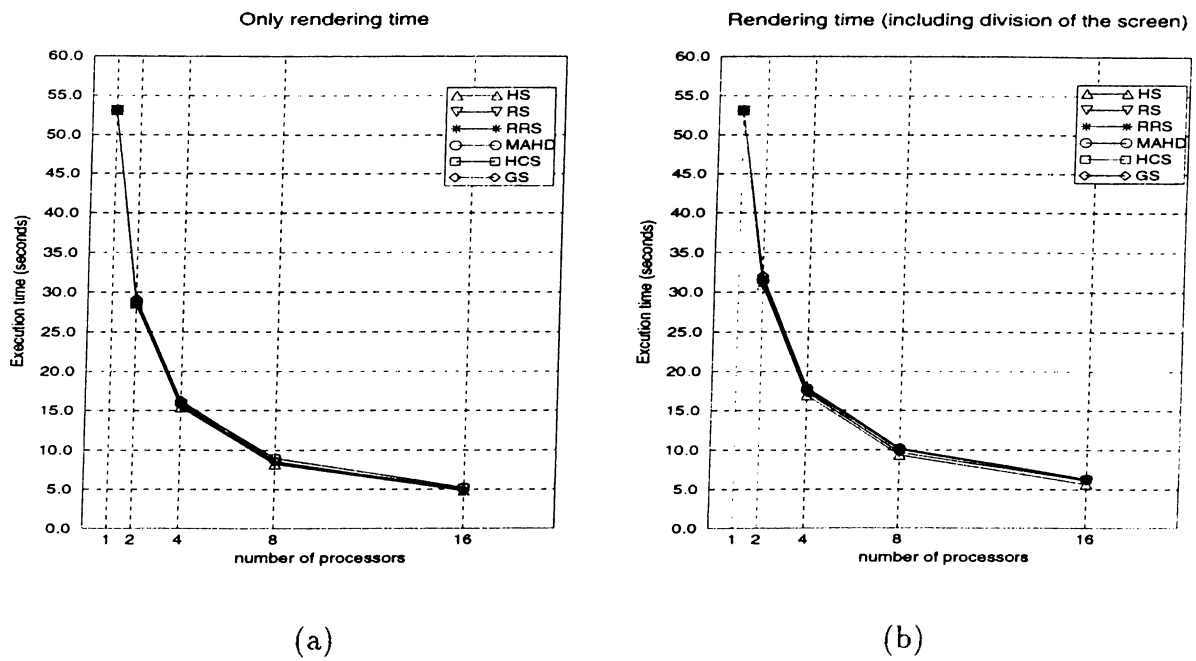


Figure 6.23: Rendering times in seconds. (a) Only rendering time excluding subdivision overhead. (b) Rendering time including subdivision overhead.

Chapter 7

Summary and Conclusions

In this thesis, we investigate utilization of distributed-memory multicomputers in three fields of computer graphics rendering: *gathering radiosity*, *polygon rendering*, and *volume rendering of unstructured grids*.

In this section, the contributions of the thesis work in each field are summarized and conclusions are presented.

7.1 Parallel Gathering Radiosity

The main issues in this work are the parallelization of form-factor matrix computation and solution phases of gathering radiosity. The contributions of the thesis are the following.

- In the form-factor computation phase, interprocessor communication is decreased by sharing the memory space between matrix elements and the objects in the scene. We propose a new demand-driven scheme to achieve better load balance in the form-factor computations. The scheme proposed in this work differs from [12, 13]. Unlike their approach, patches in the scene are not *conceptually* partitioned among processors. In this way, after matrix is calculated, matrix rows are not redistributed. However, our scheme necessitates two-level indexing in matrix-vector product operations in the solution phase. This two-level indexing is eliminated by an efficient parallel re-numbering scheme.

- Gauss-Jacobi (**GJ**) algorithm was utilized by all previous works in the solution phase. We utilize the more efficient Scaled Conjugate-Gradient (**SCG**) algorithm in the solution phase. In order to apply **SCG**, the non-symmetric coefficient matrix is converted into a symmetric matrix without perturbing the sparsity structure of the matrix.
- Parallel algorithms are developed for **GJ** and **SCG** methods for hypercube-connected multicomputers. In addition, load balancing in the solution phase is investigated. An efficient data redistribution scheme is proposed. This scheme achieves perfect load balance in matrix-vector product operations in the solution phase.

For the form-factor computation phase, experimental results indicate that it is possible to reduce the interprocessor communication by sharing the memory space for rows of the form-factor matrix with global patch data. It has been observed that demand-driven approach achieves better processor utilization in spite of its extra communication overhead.

In the solution phase, almost perfect load balance has been achieved by an efficient data redistribution scheme. This scheme brings negligible communication overhead while maintaining much better load balance during the iterations. The powerful Scaled Conjugate-Gradient method has been successfully applied in the solution phase. High efficiency values have been obtained using **SCG** with data redistribution.

We conclude that demand-driven approach is more suitable for the form-factor computation phase and **SCG** method is a much better alternative to **GJ** method in the solution phase.

7.2 Parallel Polygon Rendering

Object-space parallelism is investigated for parallel polygon rendering on hypercube-connected multicomputers. The contributions of this thesis can be summarized as follows.

- We decrease volume of communication in pixel merging phase by only exchanging local foremost pixels in each processor after local rendering phase.

- For the local rendering phase, a modified scanline z-buffer algorithm is proposed. This algorithm stores local foremost pixels into consecutive memory locations efficiently. Thus, it avoids message fragmentation while exchanging foremost pixels. In addition, initialization of scanline z-buffer, which is a sequential overhead to parallel operation, is eliminated with this algorithm.
- For pixel merging phase, we propose two schemes referred to here as *pairwise exchange* scheme and *all-to-all personalized communication* (AAPC) scheme, which are suited to the hypercube topology. Minimum number of communication steps is achieved by the pairwise exchange scheme. However, this scheme has memory-to-memory copy overhead. AAPC scheme, on the other hand, eliminates this overhead at the expense of more communication steps. Our AAPC scheme differs from 2-phase direct pixel forwarding of Lee [53]. The algorithm proposed in this work is a 1-phase algorithm, i.e., pixels are transmitted to destination processors in a single communication phase. Hence, our algorithm avoids the intermediate z-buffering in [53] totally.
- All of the processors are utilized actively throughout the pixel merging phase by exploiting the interconnection topology of hypercube and by dividing the screen among processors. We propose two heuristics, recursive subdivision and heuristic bin packing, to divide the screen adaptively for better load balancing. These heuristics utilize the distribution of foremost pixels on the screen to divide the screen.

In this work, most of the research was performed on Intel's iPSC/2 hypercube multi-computer. It is experimentally observed that exchanging only foremost pixels decreases execution time considerably. It is experimentally observed that active pixel merging with modified scanline z-buffer algorithm performs better than full z-buffer merging. The modified scanline z-buffer algorithm does not introduce much overhead to the execution. Among pixel merging schemes, all-to-all personalized communication is better than pairwise exchange scheme due to less store-and-forward overheads in spite of larger number of communication steps.

It has been observed that the *heuristic bin packing* achieves better load balance and scales better than *recursive adaptive subdivision* in active pixel merging. Therefore, it is recommended that *all-to-all personalized communication* with *heuristic bin packing* scheme should be utilized for active pixel merging on hypercube multicomputers.

Preliminary implementation of all-to-all personalized communication with heuristic bin packing on a Parsytec CC system achieves rendering rates of 300K – 700K triangles per second on 16 processors using data sets from SPD database [37]. The current implementation assumes hypercube topology and PVM is used for message passing. It is expected to achieve higher rendering rates with an implementation more suited to interconnection structure of Parsytec and using faster native message passing library.

7.3 Parallel Volume Rendering

In this work, image-space parallelism for parallel volume rendering of unstructured grids is investigated. The contributions in this thesis are the following.

- Our research focuses on the adaptive subdivision of screen for better load balance. Adaptive subdivision issue has not been investigated before in parallel volume rendering of unstructured grids. Few researchers in parallel polygon rendering [76, 99, 65, 26] and in parallel ray tracing/casting [5] investigated adaptive subdivision.
- Algorithms presented in this work can be grouped into two classes: 1-dimensional array based algorithms and 2-dimensional mesh based algorithms. Graph partitioning based subdivision and Hilbert curve based subdivision algorithms, which are mesh based algorithms, are new in parallel volume rendering field.
- The subdivision algorithms are compared experimentally on a common frame work.
- The subdivision algorithms are employed in parallelization of a volume rendering algorithm. The sequential volume rendering algorithm, based on Challinger's work [9, 10], is basically a polygon rendering based algorithm. In the previous works on parallel polygon rendering, only the number of primitives in a subregion was used to approximate the work load of the subregion. The experimental results

in our work show that the number of primitives in a region is not an enough approximation for work load. In this work, other criteria such as number of pixels and number of spans have also been utilized to approximate the work load in a region. By utilizing these additional parameters, the speedup values are almost doubled.

If the number of primitives in a region is taken as the work load of the region, the experimental results on a Parsytec CC system show that:

- Among the mesh based algorithms, graph partitioning based subdivision (GS) performs better than mesh based adaptive hierarchical decomposition (MAHD) and Hilbert curve based subdivision (HCS). Since subdivision is modeled as graph partitioning in this scheme, it has larger search space than the other algorithms.
- There exists a relation between load balancing performance of MAHD, HCS, and GS and the number of shared primitives in a region. When number of shared primitives decreases algorithms achieve better load balance. These algorithms calculate the number of primitives in a region approximately. The shared primitives cause errors in these approximate amounts. Hence, they affect the load balance.
- Among all algorithms, recursive rectangular subdivision (RRS) algorithm is the best in terms of load balance and it results in lowest number of shared primitives. The better performance of this algorithm is due to the fact that the number of primitives in a region can be calculated exactly unlike MAHD, HCS, and GS. In addition, RRS algorithm divides the screen horizontally and vertically. Thus, it has larger search space than rectangular subdivision and horizontal subdivision algorithms.

These algorithms were employed in the parallelization of a volume rendering algorithm. It has been experimentally observed that speedup values are almost doubled using additional factors such as number of pixels and number of spans in a region. Using these additional parameters, we can render the data sets used in the experiments in about 6 seconds, on the average, on 16 processors of Parsytec CC system.

Bibliography

- [1] B. Abalı, F. Özgüner, and A. Bataineh. Balanced parallel sort on hypercube multi-processors. *IEEE Trans. on Parallel and Distributed Systems*, **4**(5), 572–581 (1993).
- [2] M. B. Amin, A. Grama, and V. Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *Proceedings of 1995 Parallel Rendering Symposium*, 7–14 (October 1995).
- [3] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space filling curves and their use in the design of geometric data structures. In *2nd Inter. Symp. of Latin American Theoretical Informatics LATIN'95, Lecture Notes in Computer Science*, vol. 911, 36–48 (1995).
- [4] C. Aykanat, F. Özgüner, F. Erçal, and P. Sadayappan. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Trans. on Computers*, **37**(12), 1554–1568 (1988).
- [5] C. Aykanat, V. İşler, and B. Özgüç. Efficient parallel spatial subdivision algorithm for object-based parallel ray tracing. *Computer-Aided Design*, **26**(12), 883–890 (1994).
- [6] C. Aykanat, T. K. Çapın, and B. Özgüç. A parallel progressive radiosity algorithm based on patch data circulation. *Journal of Computers and Graphics*, **20**(2) (1996).
- [7] D. R. Baum, H.E. Rushmeier, J.M. Winget. Improving radiosity solutions through the use of analytically determined form-factors. *Computer Graphics*, **23**(3), 325–334 (1989).

- [8] A. Burke and W. Leler. Parallelism and graphics: an introduction and annotated bibliography. In *Course Notes for Siggraph Course 28, ACM Siggraph Conference*, 111–140 (1990).
- [9] J. Challinger. Parallel volume rendering for curvilinear volumes. In *Proceedings of the Scalable High Performance Computing Conference, IEEE Computer Society Press*, 14–21 (April 1992).
- [10] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *Proceedings of the 1993 Parallel Rendering Symposium, IEEE Computer Society Press*, 81–88 (October 1993).
- [11] J. Challinger. *Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids*, PhD. Thesis, University of California, Santa Cruz (1993).
- [12] A. G. Chalmers and D. J. Paddon. Implementing a radiosity method using a parallel adaptive system. In *Proceedings of the First Inter. Conf. on Appl. of Transputers* (1989).
- [13] A. G. Chalmers and D. J. Paddon. Parallel radiosity methods. In *4th North American Transputer Users Group*, Ithaca, USA (1990).
- [14] A. G. Chalmers and D. J. Paddon. Parallel processing of progressive refinement radiosity methods. In *Proc. of 2nd Eurographics Workshop on Rendering*, Barcelona, Spain (1991).
- [15] M. F. Cohen, S. Chen, J. Wallace, and D. P. Greenberg. A progressive refinement approach for fast radiosity image generation. *Computer Graphics*, **22**(4), 75–84 (1988).
- [16] M. F. Cohen and D. P. Greenberg. The Hemi-Cube : A radiosity solution for complex environments. *Computer Graphics (SIGGRAPH'85 proceedings)*, **19**(3), 31–40 (1985).

- [17] M. Cox and P. Hanrahan. Pixel merging for object-parallel rendering: A distributed snooping algorithm. In *Proceedings of the 1993 Parallel Rendering Symposium*, IEEE Computer Society Press, 49–56 (October 1993).
- [18] T. W. Crockett. Parallel rendering. *Technical Report*, ICASE Report No. 95-31, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, April 1995.
- [19] T. W. Crockett and T. Orloff. A MIMD rendering algorithm for distributed memory architectures. In *Proceedings of the 1993 Parallel Rendering Symposium*, IEEE Computer Society Press, 35–42 (October 1993).
- [20] F. C. Crow. Summed-area tables for texture mapping. *Computer Graphics*, **18**(3), 207–212 (1984).
- [21] F. C. Crow. Parallelism in rendering algorithms. In *Proceedings of Graphics Interface 88*, 87–96 (1988).
- [22] T. K. Çapın. *Parallel Processing for Progressive Refinement Radiosity*, M.S. Thesis. Bilkent University (September 1993).
- [23] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *Computer Graphics*, **22**(4), 65–74 (1988).
- [24] S. M. Drucker and P. Schroeder. Fast radiosity using a data parallel architecture. In *Proc. of 3rd Eurographics Workshop on Rendering*, Bristol, UK, 247–258 (1992).
- [25] S. Dyer and S. Whitman. A vectorized scanline z-buffer rendering algorithm. *IEEE Computer Graphics & Applications*, **7**(7), 34–45 (1987).
- [26] D. Ellsworth. A multicomputer polygon rendering algorithm for interactive applications. In *Proceedings of the 1993 Parallel Rendering Symposium*, IEEE Computer Society Press, 43–48 (October 1993).
- [27] M. Feda and W. Purgathofer. Progressive refinement radiosity on a transputer network. In *Proc. of 2nd Eurographics Workshop on Rendering*, Barcelona, Spain (1991).

- [28] T. Fröhauf. Raycasting of nonregularly structured volume data. *EUROGRAPHICS '94, Eurographics Association*, **13**(3), 293–303 (1994).
- [29] M. P. Garritty. Raytracing irregular volume data. *Computer Graphics*, **24**(5), 35–40 (1990). *Proceedings of San Diego Workshop on Volume Visualization*.
- [30] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, The MIT press, 1994.
- [31] C. Giertsen. Volume visualization of sparse irregular meshes. *IEEE Computer Graphics & Applications*, 40–48 (March 1992).
- [32] G. H. Golub and C. F. Van Loan. *Matrix computations*, 2nd Ed., The Johns Hopkins University Press (1989).
- [33] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, **18**(3), 213–222 (1984).
- [34] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. on Computers*, **c-20**(6), 623–629 (1971).
- [35] P. Guitton, J. Roman, and C. Schlick. Two parallel approaches for a progressive radiosity. In *Proc. of 2nd Eurographics Workshop on Rendering*, Barcelona, Spain (1991).
- [36] A. Gupta and A. L. Fisher. Flexible parallel polygon rendering. In *Proceedings of International Conference on Parallel Processing*, Vol.III, 87–91 (1990).
- [37] E. Haines. A proposal for standart graphics environments. *IEEE Computer Graphics & Applications*, **7**(11), 3–5 (November 1987).
- [38] B. Hendrickson and R. Leland. The Chaco user's guide (Version 1.0), Tech. Rep. SAND93-2339, Sandia National Labs. Albuquerque, NM (1993).
- [39] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Nat. Bur. Standards J. Res.*, **49**, 409–436 (1952).

- [40] J. C. Highfield and H. E. Bez. Hidden surface elimination on parallel processors. *Computer Graphics Forum*, **11**(5) 293–307 (1992).
- [41] D. Hilbert. Über die stetige Abbildung einer Linie auf Flächenstück. *Math. Annln*, **38** (1891).
- [42] V. İşler. *Spatial Subdivision for Parallel Ray Casting/Tracing*, PhD. Thesis, Bilkent University (February 1995).
- [43] J. Jájá and K. W. Ryu. Load balancing and routing on the hypercube and related networks. *Journal of Parallel and Distributed Computing*, **14**, 431–435 (1992).
- [44] J.P. Jessel, M. Paulin, and R. Caubet. An extended radiosity using parallel ray-traced specular transfers. In *Proc. of 2nd Eurographics Workshop on Rendering*, Barcelona, Spain (1991).
- [45] M. Kaddoura, C. W. Ou, and S. Ranka. Mapping unstructured computational graphs for adaptive and nonuniform computational environments. Manuscript submitted to *IEEE Trans. on Parallel and Distributed Technology* (1995).
- [46] M. Kaplan and D. P. Greenberg. Parallel processing techniques for hidden surface removal. *Computer Graphics*, **13**(2), 300–307 (1979).
- [47] G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system, Version 2.0. Dept. of Computer Science, University of Minnesota. <http://www.cs.umn.edu/~karypis>
- [48] A. Kaufman. Volume visualization. In *Volume Visualization, IEEE Computer Society Press Tutorial*, 1–18 (1990).
- [49] K. Koyamada. Fast traversal of irregular volumes. In *Visual Computing, Integrating Computer Graphics with Computer Vision*, 295–312 (1992).
- [50] T. M. Kurç, C. Aykanat, and B. Özgüç. A parallel scaled conjugate-gradient algorithm for the solution phase of gathering radiosity on hypercubes. *The Visual Computer, International Journal of Computer Graphics*, to appear (1996).

- [51] T. M. Kurç, C. Aykanat, and B. Özgüç. Active pixel merging on hypercube multicomputers. In *Lecture Notes in Computer Science*, vol. 1067, 319–326 (1996).
- [52] P. Lacroute. Real time volume rendering on shared memory multiprocessors using the shear-warp factorization. In *Proceedings of 1995 Parallel Rendering Symposium*, 15–22 (October 1995).
- [53] T. Y. Lee, C. S. Raghavendra, and J. B. Nicholas. Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers. *IEEE Trans. on Visualization and Computer Graphics*, **2**(3), 202–217 (1996).
- [54] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, **8**(3), 29–37 (1988).
- [55] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, **9**(3), 245–261 (1990).
- [56] J. Li and S. Miguet. Z-buffer on a transputer-based machine. In *Proceedings of the Sixth Distributed Memory Computing Conf.*, IEEE Computer Society Press, 315–322 (April 1991).
- [57] B. Lucas. A scientific visualization renderer. In *Proceedings of IEEE Visualization '92*, IEEE Computer Society Press, 227–234 (October 1992).
- [58] K. Ma and J. S. Painter. Parallel volume visualization on workstations. *Computers & Graphics*, **17**(1), 31–37 (1993).
- [59] K. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, **14**(4), 59–67 (1994).
- [60] K. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory multicomputers. In *Proceedings of 1995 Parallel Rendering Symposium*, 23–30 (October 1995).

- [61] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, **14**(4), 23–32 (July 1994).
- [62] S. Molnar, J. Eyles, J. Poulton. PixelFlow: high-speed rendering using image composition. *Computer Graphics*, **26**(2), 231–240 (1992).
- [63] C. Montani, R. Perego, and R. Scopigno. Parallel rendering of volumetric data set on distributed-memory architectures. *Concurrency: Practice and Experience*, **5**(2), 153–167 (1993).
- [64] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of hilbert space-filling curve. Technical report UMCP-CSD:CS-TR-3611, Umiacs, University of Maryland (1996).
- [65] C. Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, 75–84 (1995).
- [66] S. F. Nugent. The iPSC/2 direct-connect communications technology. In *Proceedings of Third Conf. Hypercube Concurrent Comput. and Appl.*, 51–60 (January 1988).
- [67] D. J. Paddon, A. Chalmers, and D. Stuttard. Multiprocessor models for the radiosity method. In *Proc. of the First Bilkent Computer Graphics Conference on Advanced Techniques in Animation, Rendering, and Visualization*, B. Özgüç and V. Akman (Eds.), Ankara, 85–103 (1993).
- [68] B. T. Phong. Illumination for computer generated pictures. In *Tutorial: Computer Graphics, 2nd edition, Graphics and Image Processing*, Computer Society Press, 449–455 (1982). Reprinted from *Communications of ACM*, **18**(6), 311–317 (1974).
- [69] J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Trans. on Parallel and Distributed Systems*, **7**(3), 288–299 (1996).
- [70] J. Pineda. A parallel algorithm for polygon rasterization. *Computer Graphics*, **22**(4), 17–20 (1988).

- [71] C. G. Plaxton. Load balancing, selection and sorting on the hypercube. In *Proceedings of 1989 ACM Symp. Parallel Algorithms and Architectures*, 64–73 (1989).
- [72] PowerPVM/EPX for Parsytec CC systems: PowerPVM/EPX User's Guide, *Genias Software GmbH*.
- [73] M. Price and G. Truman. Radiosity in parallel. *Applications of Transputers*, IOS Press, Washington (Proc. of the First International Conf. on App. of Transputers, Aug. 1989), 40-47 (1990).
- [74] W. Purgathofer and M. Zeiller. Fast radiosity by parallelization. In *Proceedings of the Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, Rennes, 173-184 (1990).
- [75] S. Ranka and S. Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition*, Bilkent University Lecture Series, Springer-Verlag (1990).
- [76] D. R. Roble. A load balanced parallel scanline z-buffer algorithm for the iPSC hypercube. In *Proceedings of Pixim' 88*, Paris, 177–192 (October 1988).
- [77] D. F. Rogers. *Procedural Elements for Computer Graphics*, McGraw-Hill (1985).
- [78] K. W. Ryu and J. Jájá. Efficient algorithms for list ranking and for solving graph problems on the hypercube. *IEEE Trans. on Parallel and Distributed Systems*, **1**(1), 83–90 (1990).
- [79] Y. Saad and M. H. Schultz. Topological properties of hypercubes. *Research Report YALEU/DCS/RR-389* (June 1985).
- [80] P. Sabella. A rendering algorithm for visualizing 3D scalar fields. *Computer Graphics*, **22**(4), 51–58 (1988).
- [81] R. Scopigno, A. Paoluzzi, S. Guerrini, and G. Rumolo. Parallel depth-merge: A paradigm for hidden surface removal. *Computers & Graphics*, **17**(5), 583–592 (1993).

- [82] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, **24**(5), 63–70 (1990). *Proceedings of San Diego Workshop on Volume Visualization*.
- [83] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing 93*, 54–65 (1993).
- [84] D. Speray and S. Kennon. Volume Probes: interactive data exploration on arbitrary grids. *Computer Graphics*, **24**(5), 5–12 (1990). *Proceedings of San Diego Workshop on Volume Visualization*.
- [85] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. In *Tutorial: Computer Graphics, 2nd edition, Graphics and Image Processing*, Computer Society Press, 270–280 (1982). Reprinted from *Communications of ACM*, **17**(1) (1974).
- [86] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, **6**(1), 1–55 (1974).
- [87] A. State, J. McAllister, U. Neumann, H. Chen, T. J. Cullip, D. T. Chen, and H. Fuchs. Interactive volume visualization on a heterogeneous message-passing multicomputer. In *1995 Symposium on Interactive 3D Graphics*, 69–74 (1995).
- [88] B. Tabatabai, E. A. Sessarego, and H. F. Mayer. Volume rendering on non-regular grids. In *Proceedings of EUROGRAPHICS '94, Eurographics Association*, **13**(3), 248–258 (1994).
- [89] E. Tanin, T. M. Kurç, C. Aykanat, and B. Özgüç. Comparison of two image-space subdivision algorithms for direct volume rendering on distributed-memory multicomputers. In *Lecture Notes in Computer Science*, vol. 1041, 503–512 (August 1995).
- [90] E. Tanin. *Comparison of Image Space Subdivision Algorithms for Parallel Volume Rendering*, M.S. Thesis, Dept. of Computer Engineering and Information Sci., Bilkent University (July 1995).

- [91] T. Theoharis and I. Page. Parallel incremental polygon rendering on a SIMD processor array. In *Parallel Processing for Computer Vision and Display*, Eds. P. M. Drew, T. R. Heywood, and R. A. Earnshaw, Addison-Wesley, 329–337 (1989).
- [92] C. Upson and M. Keeler. VBUFFER: Visible volume rendering. *Computer Graphics*, **22**(4), 59–64 (1988).
- [93] A. Van Gelder and J. Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. *Technical Report*, UCSC-CRL-93-02, Computer and Info. Sciences, University of California, Santa Cruz (1993).
- [94] A. Varshney and J. F. Prins. An environment-projection approach to radiosity for mesh-connected computers. In *Proc. of 3rd Eurographics Workshop on Rendering*, Bristol, UK, 271–281 (1992).
- [95] A. Watt. *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley (1989).
- [96] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques, theory and practice*, Addison-Wesley (1992).
- [97] R. Weinberg. Parallel processing image synthesis and anti-aliasing. *Computer Graphics*, **15**(3), 55–62 (1981).
- [98] L. Westover. Footprint evaluation for volume rendering. *Computer Graphics*, **24**(4), 367–376 (1990).
- [99] S. Whitman. *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers (1992).
- [100] S. Whitman. Computer graphics rendering on a parallel processor. In *Course Notes for Siggraph '90, Course 28*, 167–183 (1990).
- [101] S. Whitman and R. Parent. A survey of parallel hidden surface removal algorithms. In *Proceedings of Pixim' 88*, Paris (1988).

- [102] T. Whitted. An improved illumination model for shaded display. In *Graphics and Image Processing, Communications of the ACM*, **26**(6), 342–349 (1980).
- [103] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. *Computer Graphics*, **25**(4), 275–284 (1991).
- [104] P. L. Williams. Visibility ordering meshed polyhedra. *ACM Trans. on Graphics*, **11**(2), 103–126 (1992).
- [105] P. L. Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*, PhD. Thesis, University of Illinois at Urbana-Champaign (1992).
- [106] R. Yagel. Volume viewing: state of the art survey. In *Visualization '93, Tutorial #9, Course Notes: Volume Visualization Algorithms and Applications*, 82–102 (1993).
- [107] R. Yagel and R. Machiraju. Data-parallel volume rendering algorithms. *The Visual Computer*, **11**, 319–338 (1995).