PARALLEL PROCESSING FOR PROGRESSIVE REFINEMENT RADIOSITY

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Tolga K. ÇAPIN
September, 1993

PARALLEL PROCESSING FOR PROGRESSIVE REFINEMENT RADIOSITY

A THESIS

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

MASTER OF SCIENCE

By

Tolga K. Çapın

September, 1993

Tolga K. ÇAPIN

[illegible]

TA
1632
C37
1993

B. 4213

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

C. Aykanat

Asst. Prof. Cevdet Aykanat (Advisor)

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

[illegible]

Prof. Bülent Özgüç

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

Fazlı Can

Vis. Assoc. Prof. Fazlı Can

Approved for the Institute of Engineering and Science:

M. Baray

Prof. Mehmet Baray
Director of the Institute

ABSTRACT

PARALLEL PROCESSING FOR PROGRESSIVE REFINEMENT RADIOSITY

Tolga K. Çapın
M.S. in Computer Engineering and Information Science
Advisor: Asst. Prof. Cevdet Aykanat
September, 1993

Progressive refinement radiosity is an increasingly popular method for realistic image synthesis of non-existing environments. The method successfully approximates the light distribution in an environment, however it requires excessive amount of computation. In this thesis, the progressive refinement method is investigated for parallelization on ring and hypercube-connected multicomputers. Two different approaches for parallelization, based on synchronous parallelism with static task assignment, are proposed, in order to achieve better coherence in parallel light distributions and obtain good performance on simple topologies. Efficient global circulation schemes are proposed in order to decrease the total volume of communication by asymptotical factors. The first scheme for parallelization is a modification of the sequential algorithm in that several patches shoot their energy at a time, while the second scheme is based on the parallelism level of one shooting patch at a time. The proposed parallel algorithms are evaluated theoretically and implemented for ring and hypercube-connected topologies on Intel's iPSC/2 multicomputer. Load balance quality of the proposed schemes are evaluated experimentally.

Keywords: Realistic Image Synthesis, Parallel Computing, Multicomputers, Radiosity, Progressive Refinement Radiosity, Ring Interconnection Topology, Hypercube Interconnection Topology.

ÖZET

DERECELİ GELİŞEN IŞIMA İÇİN PARALEL İŞLEME

Tolga K. Çapın
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Danışman: Yrd. Doç. Dr. Cevdet Aykanat
Eylül, 1993

Dereceli gelişen ışıma gerçeğe uygun görüntü üretmek için gittikçe daha fazla kullanılmakta olan bir yöntemdir. Yöntem, ışığın sahnede dağılımını başarılı bir şekilde hesaplamakta, ancak çok fazla işlem gerektirmektedir. Bu tezde, dereceli gelişen ışıma yönteminin zincir ve hiperküp bağlantılı dağıtık hafızalı çok işlemciler üzerinde paralel hesaplanması araştırılmaktadır. Işık dağılımının sırasının sağlanması ve basit topolojilerde iyi performans sağlanabilmesi için eşzamanlı paralel işlemeye dayalı iki yaklaşım geliştirilmiştir. Toplam iletişim miktarını asimtotik olarak azaltmak için verimli dolaştırma yöntemleri önerilmiştir. Önerilen ilk paralel yaklaşım, tek-işlemcili algoritmaya değişiklik getirmiştir, çünkü bu yaklaşımda aynı anda birden fazla yüzey ışık yayar. İkinci yaklaşım aynı anda sadece bir yüzey yayıcı yöntemine göre tasarlanmıştır. Önerilen yöntemler zincir ve hiper-küp bağlantılı dağıtık hafızalı çok işlemciler için hiperküp bağlantılı Intel iPSC/2 bilgisayarında gerçekleştirilmiştir. Önerilen yöntemlerin iş dağılımı kalitesi deneysel olarak gözlenmiştir.

Anahtar Sözcükler: Gerçeğe Uygun Görüntü, Üretme, Paralel İşleme, Dağıtık Hafızalı Çok İşlemciler, Işıma, Dereceli Gelişen Işıma, Zincir Bağlantılı Topoloji, Hiperküp Bağlantılı Topoloji.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my supervisor Asst. Prof. Cevdet Aykanat for his guidance, suggestions, and invaluable encouragement throughout the development of this thesis.

I would also like to thank Prof. Bülent Özgüç and Assoc. Prof. Fazlı Can for reading and commenting on the thesis.

I would like to acknowledge Guy Moreillon for his house interior model, Bilge Erkan for her glass model, and Aydın Ramazanoğlu for photographing the final images.

I am grateful to the members of my family and Meltem for their infinite moral support and patience that they have shown, particularly in times I was not with them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Realistic image synthesis of fictitious environments is one of the major problems of computer graphics, and has a wide range of applications such as animation, CAD, advertising, scientific applications. In order to have photorealistic image quality, the global illumination of the input environment (indirect lighting, surface-to-surface interreflections and shadows) must be simulated accurately. The original radiosity method is one of the successful solutions to this problem. This method requires excessive time and memory, and the *progressive refinement* approach for radiosity has been proposed to provide fast computation rate by approximating the original method. However this method still requires excessive computational power and requires acceleration techniques such as exploiting parallelism. This thesis will examine the parallelization of progressive refinement radiosity for medium-to-coarse grain multicomputers.

The earlier, but still used, methods for rendering, Gouraud and Phong shading methods [9, 23], assume that objects are illuminated directly by point light sources located at infinity. Although these methods support specular and diffuse illumination; they are local methods, since they ignore the indirect illumination by surface-to-surface interreflections and shadows caused by occlusions among the objects. Hence these methods produce incorrect simulations of the light distribution. Ray tracing [44] has become the preferred method for environments consisting of mainly specular surfaces to give solutions that are view-dependent. Hence, the ray tracing method requires recomputation of light distributions each time the viewing position changes. The radiosity method can successfully compute the interreflections among diffuse surfaces, which are approximated by a constant "ambient" term in ray tracing. The radiosity method is distinct among the other methods such that it separates the light calculations from the rendering process; once the color (illumination) of each object is calculated, one can walk through the environment without requiring further color calculations.

The *progressive refinement radiosity* method [16] reformulates the original radiosity method and aims to provide approximated solutions initially, in near interactive time. However, the method still requires excessive time in order to provide successful images for complex scenes. Hence, several methods have been proposed in order to speed up the algorithm further. Exploiting parallelism is among the methods that can achieve good results.

1.2 Progressive Refinement Radiosity

The conventional radiosity method is based on radiative heat transfer and was introduced to the field of Computer Graphics by Goral *et al* [22]. The method is based on an energy equilibrium within an enclosure. The objects are given as input with their positions in 3D, their reflectivity and emission values for each colorband. The reflectivity of an object gives the fraction of the incident (impinging) light it disperses and the emission of an object is defined as the amount of light it radiates. The reflectivity and emission values are predetermined which are constants and they depend only on the surface characteristics of the objects. The radiosity method aims to find the *radiosity* of each surface element; that is the amount of light energy leaving each surface, which is the sum of emission from the surface and reflection of the incident light (that is the light arriving) at the surface. The incident light is determined by the effects of all other surfaces on the surface, which depends on the geometrical factors as well as the radiosity of other surfaces.

The main equation for the radiosity method is:

$$\begin{aligned} Radiosity_i \times Area_i &= Emission_i \times Area_i + Reflectance_i \times \\ &\times \sum_j Radiosity_j \times Area_j \times FormFactors_{ji} \end{aligned} \quad (1.1)$$

This equation states that the light leaving from a surface is the sum of light originally emitted from the surface and reflection of the incident light.

The emission (the first term on the right hand side of Eq. 1.1) is equal to 0 if the surface is not a light source. The second term in Eq. 1.1 corresponds to the reflection of incident light. Here, $FormFactor_{ji}$ denotes the fraction of light energy leaving surface j that lands on surface i, to all the energy leaving surface j. Note that the radiosity of a surface is dependent on the radiosity of all other surfaces, and Eq. 1.1 holds for each surface; hence combining the equations for all surfaces, a linear system of equations is formed for solving the radiosities of all surfaces, after which the final image can be rendered.

The conventional radiosity method, described above, is expensive in terms of execution and the final image cannot be viewed until the matrix equation is solved. This reduces the usability of the method for complex scenes consisting of large number of surfaces, unless the *progressive refinement approach* is used. This new approach eliminates setting up the system of equations, and follows the path the light travels in the environment. The initially approximated solutions are provided quickly and the final solution is approached iteratively. At each iteration:

1. The most energetic surface is selected as the source surface,
2. The form-factors from this source surface are computed,
3. Depending on these form-factors, the light is distributed from the source surface to the environment.

In this method, intermediate solutions between the iterations can be viewed, thus allowing the modification of the input scene geometry and surface color properties without waiting until the complete solution is achieved.

1.3 Motivation

Current research in radiosity has concentrated on two main classes: increasing the accuracy and the speed of the solution. The excessive amount of computation required by the conventional radiosity led to research for speeding up the solution. Although algorithmic and meshing techniques decrease the execution time, still excessive computational power is required. Hence, exploiting parallelism can be used for speeding up the method further. However, there is not much research on parallelization of radiosity. This thesis aims to fill this gap by providing a thorough examination of parallelism in progressive refinement radiosity, and presents two different approaches to parallelization of progressive refinement radiosity on multicomputers. Synchronous parallelism with static task assignment is exploited in order to obtain a solution whose performance does not degrade with simple topologies while exploiting the shooting patch coherence. The two schemes are implemented for ring and hypercube-connected multicomputers on the Intel's iPSC/2 multicomputer. The performances of the algorithms are analyzed both theoretically and experimentally.

1.4 Outline of the Thesis

This thesis includes three main parts. In the first part, background for the theory of progressive refinement radiosity is provided. The second part discusses the design criteria for developing parallel algorithms and previous work on parallelization of the radiosity method. The third part includes two considerably different approaches for parallel progressive refinement radiosity, the approaches are presented and the theoretical and experimental evaluation of the algorithms are given. Chapter 2 contains the radiosity background. Chapter 3 contains the parallelization issues for the method. Chapters 4 and 5 present the two novel approaches and give the performance analysis. Chapter 6 includes the concluding remarks, and the discussion of future work directions.

Chapter 2

Radiosity

In this chapter, the realistic image generation problem is stated, the difference between local and global illumination is discussed, and the radiosity method is explained. The chapter continues with the discussion of *progressive refinement radiosity*, which is investigated for parallelization in this thesis.

2.1 Realistic Image Generation

In realistic image generation, the input is a set of objects in 3D with their light characteristics. The output is a photographic quality image of the fictitious scene.

The image synthesis process consists of the following phases:

1. Read in object data from disk.
2. Determine the illumination distribution in the environment and find the color of each object or surface.
3. Render the surface data onto the image and display the image.

The objects are generally approximated by polygons. The input dataset consists of the *(x,y,z)* coordinates and id's of the polygons in the scene, as well as the reflectivity and the emission of the polygons for color-bands *(red,green,blue)*. The polygons are assumed to be planar. The final image is a two-dimensional array of pixels for three color-bands. The display of the final image depends on the viewing position and direction of the viewer in three-dimensional coordinates, and the display process consists of geometrical transformations from the world coordinates to the viewing coordinate system, clipping of the objects to eliminate invisible parts of the objects from the viewing position, and accurate displaying of the objects according to their computed color and geometry.

The method for computing the illumination should be selected carefully in order to get photographical quality images. The illumination models used in this phase simulate the physical phenomena; and the shadows, which contribute to the quality of the final image are computed.

Earlier, however still in use, illumination methods [9, 23] assume the input objects independently and accept the light sources at infinity in order to compute the illumination efficiently. In these methods, the color information [23] or the normal vectors [9] are interpolated in order to have smooth shading of the polygons for realistic image generation. However, these methods are *local*, that is they consider only direct illumination from a light source and ignore object-to-object illuminations and shadows caused by occlusions among the objects.

Two main global approaches that have been popular and successful at generating photographic quality images are *ray tracing* and *radiosity*. In ray tracing, for each pixel on the image plane, a ray is shot and the ray is traversed by reflections and refractions among the objects in the scene. Ray tracing is suitable for scenes consisting of mainly specular surfaces. On the other hand, the radiosity method is successful for environments with diffuse surfaces. The radiosity method is based on an energy equilibrium within a closed environment.

In ray tracing, the ambient light (i.e. the indirect light resulting from diffuse reflections among the objects) is approximated by a constant factor, however the radiosity method can compute the ambient term more accurately. Another advantage of radiosity over ray tracing is: in an environment with diffuse objects, once the light distribution is computed, one can walk through the scene in near-interactive time with no further light distribution computations, provided that the geometry does not change.

The radiosity method has a major difference than the other methods: it seperates the light distribution computation process from the display process. For example, in ray tracing, the light distribution is computed for each image, as the rays are traversed in the environment. This property makes the radiosity method an efficient method for realistic image synthesis, especially for walkthroughs.

2.2 The Radiosity Method

The radiosity method, based on an energy equilibrium within an enclosure, has its basis in heat transfer between surfaces in an environment [35]. The method was introduced to the field of Computer Graphics by Goral *et al.* [22]. In Computer Graphics, the energy that is transferred is the light energy compared to the heat energy examined by Thermodynamics.

The radiosity method assumes perfect Lambertian surfaces (i.e. they emit or reflect light in all directions with equal intensity). Given the objects (surfaces) in the scene with their emissions and reflectivities for three color-bands red, green, blue; the light distribution is formulated based on the surface characteristics and geometrical relationships. The following equation, which states that light leaving a surface is the sum of self-emitted energy (if the surface belongs to a light-emitter object) and reflection of the incident light incoming from all other surfaces, holds for each surface i in the environment:

$$B_i = E_i A_i + r_i \int_{env\ j} B_j F_{ji} dA_j \tag{2.1}$$

where,

Radiosity(B) : The total rate of energy leaving one surface.
(energy/unit area)

Emission(E) : Rate of energy (light) emitted from a surface.
(energy/unit area)

Reflectivity(r) : Fraction of light reflected back to the environment.
(unitless)

Form Factor(F_{ji}) : Fraction of light leaving surface j which lands on i, to all light leaving j. (unitless)

Area(A) : Area of surface (unit area)

The integral in Equation 2.1 is inefficient to evaluate for differential areas, therefore all the input surfaces in the environment are subdivided into smaller *patches*, which are assumed to have constant radiosity (energy). Then, the following equation holds for each patch i in the environment:

$$B_i A_i = E_i A_i + r_i \sum_{j=1}^{N} B_j A_j F_{ji}, \quad 1 \leq i \leq N \tag{2.2}$$

Radiosity(B) : The total rate of energy leaving one patch.
(energy/unit time/unit area)

Emission(E) : Rate of energy (light) emitted from a patch.
(energy/unit time/unit area)

Reflectivity(r) : Fraction of light reflected back to the environment.
(unitless)

Form Factor(F) : Fraction of light leaving one patch which lands on another.
(unitless)

N : Number of patches in the environment.

Note that the following reciprocity equation holds for each patch pair:

$$A_j F_{ji} = A_i F_{ij} \tag{2.3}$$

So, Equation 2.2 can be rewritten by using Eq. 2.3 and eliminating A_i's as:

$$B_i = E_i + r_i \sum_{j=1}^{N} B_j F_{ij}, \quad 1 \leq i \leq N \tag{2.4}$$

Combining Eq. 2.4 for all patches in the environment yields following linear system of equations:

$$\begin{bmatrix} 1-r_1F_{1,1} & -r_1F_{1,2} & \cdots & -r_1F_{1,N} \\ -r_2F_{2,1} & 1-r_2F_{2,2} & \cdots & -r_2F_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ -r_NF_{N,1} & -r_NF_{N,2} & \cdots & 1-r_NF_{N,N} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{bmatrix} \tag{2.5}$$

Equation 2.5 has to be solved for B_i 's in three color-bands. The reflectivity r_i and emission E_i values of the patches are constants and are determined by only the characteristics of the objects which are approximated by patches. In order to solve Eq. 2.5, the form-factor values (F_{ij}'s) should be computed. The form-factor value for a patch depends only on geometrical relationships among the patches and is discussed in the following section.

2.2.1 Form-Factor Definition

The form-factor is the fraction of the energy leaving one patch which lands onto another patch, to all the energy leaving the first patch [14, 22, 35]. By definition, the sum of all the form-factors from a patch is equal to unity. The form-factor from a planar or convex patch to itself is zero.

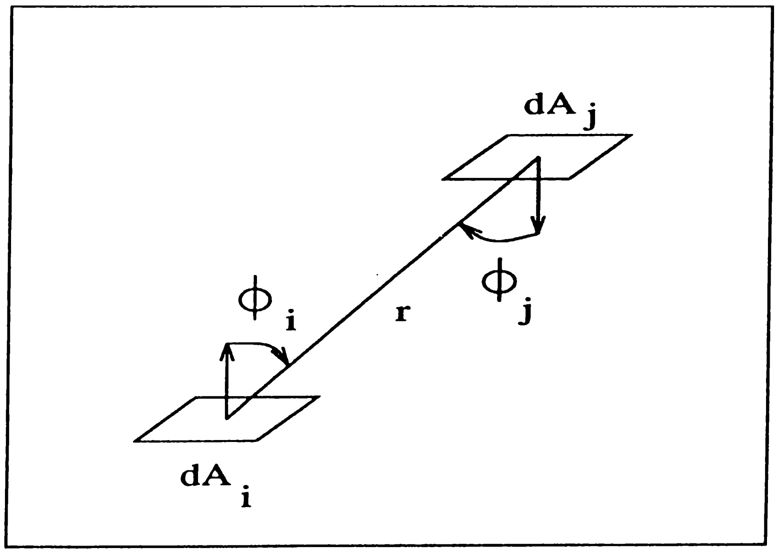


Figure 2.1. Form-Factor Geometry

Figure 2.1 shows the geometric relationships for form-factor computation. In this figure dA_i and dA_j correspond to differential area elements of patch i

and j, respectively. Then, the form-factor from differential area i to differential area j is given by:

$$F_{dA_i dA_j} = \frac{cos\Phi_i cos\Phi_j}{\pi r^2_{dA_i dA_j}} \tag{2.6}$$

Integrating over patch j, the form-factor from differential area dA_i to area j is given as:

$$F_{dA_i A_j} = \int_{A_j} \frac{cos\Phi_i cos\Phi_j}{\pi r^2_{dA_i dA_j}} dA_j \tag{2.7}$$

The form-factor between finite-area patches is defined as the area average of the differential-to-finite area form-factors:

$$F_{A_i A_j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\Phi_i cos\Phi_j}{\pi r^2_{dA_i dA_j}} dA_j dA_i \tag{2.8}$$

Almost always, scenes have occlusions, i.e. some part of the patch is not visible from a second patch because of a third patch between them. Then, a function $\delta_{dA_i dA_j}$ is needed to specify whether the differential area dA_i "sees" differential area dA_j:

$$F_{A_i A_j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\Phi_i cos\Phi_j}{\pi r^2_{dA_i dA_j}} \delta_{dA_i dA_j} dA_j dA_i \tag{2.9}$$

In Eq. 2.9, $\delta_{dA_i dA_j}$ takes value 1 or 0 depending on the visibility between two differential areas dA_i and dA_j.

2.2.2 Form-Factor Computation: Hemicube Method

The hemicube method [14] is proposed in order to provide an efficient, but approximated solution to form-factor calculation and it handles occlusions among the patches in the environment. The following subsections discuss the hemicube method in detail.

Approximations of the Hemicube Method

If the distance between two patches is considerably large compared to their size and the effect of occlusion is assumed negligible, the value of the inner integral in Eq. 2.9 remains almost constant because Φ_i, Φ_j, and r change slightly for differential areas of patch i and j. In this case, the effect of the outer integral is multiplication by unity and the patch-to-patch form-factor is reduced to:

$$F_{A_iA_j} \approx F_{dA_iA_j} = \int_{A_j} \frac{cos\Phi_i cos\Phi_j}{\pi r^2} dA_j \tag{2.10}$$

The source differential area (dA_i) is selected as the center point of the patch i in order to represent a well-approximated average position for patch i.

The hemicube method is based on the geometric analogue developed by Nusselt [35]. The form-factor is equivalent to the fraction of the circle (which is the base of the hemisphere placed over the source patch) that is covered by projecting the destination patch onto the hemisphere and orthographically onto the circle. Figure 2.2 illustrates the geometry of Nusselt's Analogue. Each point on the circle has an associated delta form-factor, hence the form-factor of a patch is computed by adding the delta form-factors of the points in the projection area.

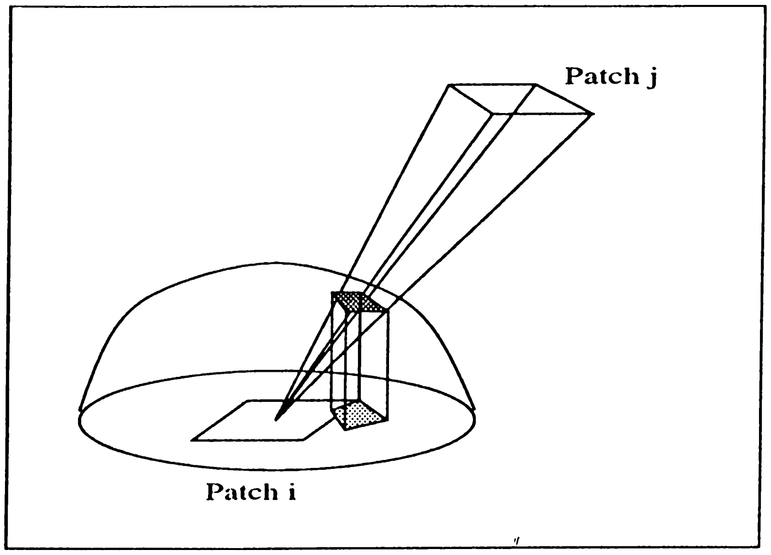


Figure 2.2. Nusselt's Analogue

However, discretizing the hemisphere requires creating equal-sized elements on the hemisphere as well as setting up a set of linear coordinates to uniquely describe locations on the hemisphere surface make the hemisphere method impractical; therefore the hemicube method is used as an approximation of the hemisphere. The hemicube method provides an efficient solution to the form-factor computation for general complex scenes. The method can also be used on special hardware designed for z-buffer hidden surface removal generally used for rendering.

Summary of the Hemicube Method

In the hemicube method, instead of projecting onto a hemisphere, a hemicube is placed onto the center of the source patch (Figure 2.3). Then the environment is transformed to the viewing coordinates of the source patch so that the center of the source patch is at the origin and the normal vector of the source patch coincides with the +y axis. Hence, five faces of the hemicube (the top face, facing +y axis, four side faces facing -x, +x, -z, +z axes) replace the hemisphere. These hemicube faces are divided into small square "pixels" at a given resolution (generally 50x100x100), and the environment is projected and filled onto the five planar faces. If more than one patch project onto the same pixel, the nearest one is selected as the visible patch. This necessitates holding an item-buffer for the hemicube. The item-buffer holds, for each pixel, the id of the patch with nearest distance to the center of the source patch and its distance. This process is similar to the *z-buffer* hidden-surface removal [39].

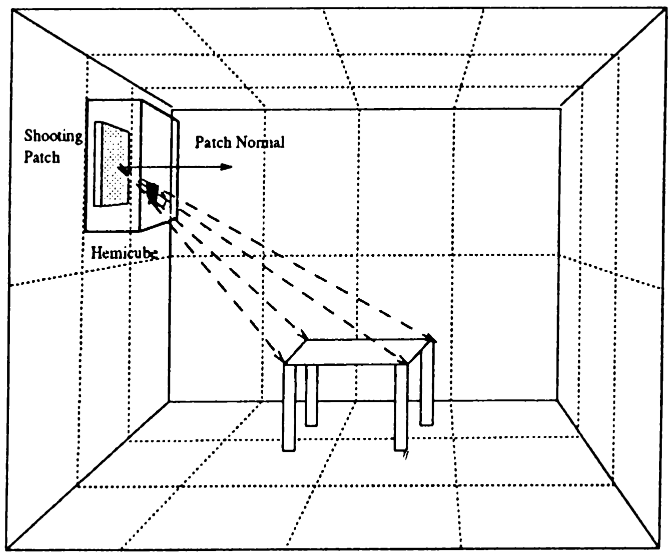
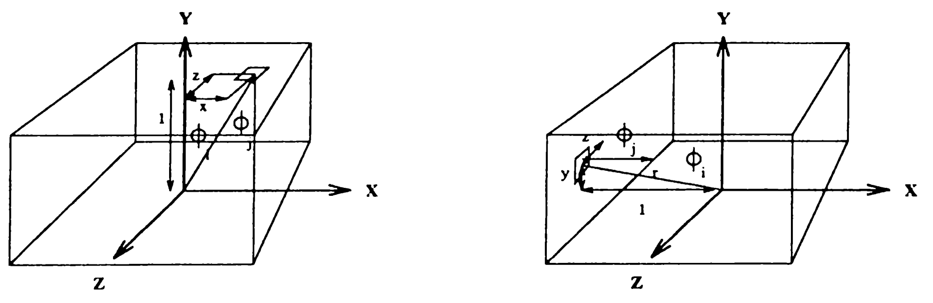


Figure 2.3. The Hemicube Method

Having projected all the environment onto the item-buffers, the buffer entries are converted to a form-factor vector ($F_{ij}, 1 \leq j \leq N$) corresponding to the source patch i. This process consists of adding the delta form-factors of the pixels that correspond to the same patch j to compute the form-factor value F_{ij}. The delta form-factors are computed using the approximation stated in Equation 2.10.

The delta form-factor derivation for a pixel on the top face and on a side face are given in Figure 2.4.



TOP OF HEMICUBE

$r = \sqrt{x^2 + z^2 + 1}$

$\cos \Phi_i = \cos \Phi_j$

$\cos \Phi = \dfrac{1}{r}$

$$\Delta \text{ Form-Factor} = \frac{\cos \Phi_i \cos \Phi_j}{\Pi r^2} = \frac{1}{\Pi (x^2 + z^2 + 1)}$$

SIDE OF HEMICUBE

$r = \sqrt{y^2 + z^2 + 1}$

$\cos \Phi_i = \dfrac{y}{r}$

$\cos \Phi_j = \dfrac{1}{r}$

$$\Delta \text{ Form-Factor} = \frac{\cos \Phi_i \cos \Phi_j}{\Pi r^2} = \frac{y}{\Pi (y^2 + z^2 + 1)}$$

Figure 2.4. Delta Form-Factor Derivation

Detailed Description of the Hemicube Method

The projection of other patches in the environment onto the hemicube requires passing all patches through a *projection pipeline* consisting of *a)* visibility test, *b)* clipping, *c)* perspective projection, and *d)* scan-conversion onto the hemicube faces.

For each destination patch j in the environment, the following is performed:

a) Visibility test: First, it has to be checked whether the destination patch j is potentially visible from the source patch. If patch j is behind the source patch i or they do not face each other, then patch j requires no further computations and leaves the projection pipeline in the early stage.

Each planar patch has only one face and a constant normal vector. This normal is computed at the initialization phase of the patch data. Assuming the vertices of the triangular patch are input in clockwise manner, the normal vector of the patch is found as:

$$\mathbf{N} = (\mathbf{V}_3 - \mathbf{V}_1) \times (\mathbf{V}_2 - \mathbf{V}_1) \tag{2.11}$$

where "-" and "×" are vector subtraction and vector cross product. Figure 2.5 illustrates the geometry of the patch normal vector computation.

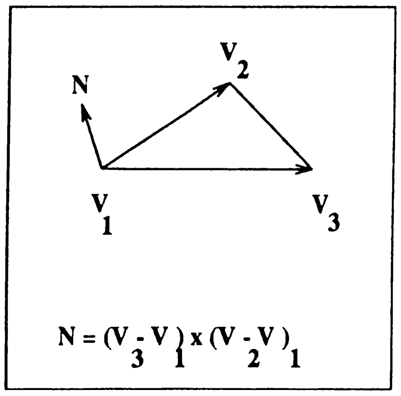


Figure 2.5. Normal Vector Computation

The visibility test consists of two phases. In the first phase, it is tested whether the two patches face each other. This is accomplished by evaluating the formula:

$$\mathbf{N_j} \cdot (\mathbf{S_j} - \mathbf{S_i}) \leq 0 \tag{2.12}$$

where $\mathbf{N_j}$ is the normal vector of patch j, $\mathbf{S_i}$ and $\mathbf{S_j}$ are the centers of patch i and j, and $(\mathbf{S_j} - \mathbf{S_i})$ is the vector connecting the two patch centers which has the starting point at the center point of patch i. If the dot product "·" in Eq. 2.12 is negative, the patch is potentially visible and the visibility test continues. If the dot product is positive, the patches do not face each other and patch j leaves the projection pipeline.

If the first phase is successful. it is checked whether the destination patch is situated above the source patch plane. This is accomplished by evaluating:

$$\mathbf{N_i} \cdot (\mathbf{V_1^j} - \mathbf{S_i}) \geq 0 \text{ or } \mathbf{N_i} \cdot (\mathbf{V_2^j} - \mathbf{S_i}) \geq 0 \text{ or } \mathbf{N_i} \cdot (\mathbf{V_3^j} - \mathbf{S_i}) \geq 0 \qquad (2.13)$$

where $(\mathbf{V_1^j}, \mathbf{V_2^j}, \mathbf{V_3^j})$ are the three vertices of the triangular patch j. If Equation 2.13 is true, then at least one vertex of the destination patch j is above the source patch, so patch j is potentially visible. Otherwise, patch j is below the source patch plane, therefore it is not visible, so patch j leaves the projection pipeline.

b) Viewing transformation and clipping: The patch that has passed the visibility test is potentially visible. Next, the geometrical transformations have to be performed in order to bring the destination patch j to the viewing coordinate system. This is done by setting up the viewing matrix to transform the world coordinated system to the viewing coordinate system for the source patch i initially and multiplying destination patches j in the pipeline with this matrix.

When the destination patch j has been brought to the viewing coordinate system, it will be projected onto the five faces of the hemicube. If R is the half-width resolution of the hemicube, then a top face with resolution 2Rx2R facing +y direction and four side faces of resolution 2RxR facing -x, +x, -z, +z directions are placed. This results in clipping planes $Y = X, Y = -X, Y = Z, Y = -Z, Z = -X, Z = X, Y = 0$.

When projecting the patch onto a hemicube face, parts of the destination patch j which are invisible from patch i through the hemicube face should be clipped with respect to the surrounding clipping planes of the face. So, the output is a set of vertex points which form the intersection polygon between the input patch j and the clipping volume defined by four clipping planes. Sutherland-Hodgman [38] polygon clipping algorithm is used in this thesis.

c) Perspective Projection: After the destination patch has been clipped with respect to the viewing volume for a hemicube face, only the part of the patch which is inside the volume has been left. Next, perspective projection with focus length equal to 1 of the vertices is performed to have their coordinates on the hemicube face.

d)Scan-conversion: Having computed the locations of the vertex projections on the hemicube face, the next process is to "fill" the hemicube face with this patch, considering the hidden surface elimination. First, an edge list corresponding to the points on the edges of the patch is created. Then, the two corresponding edge points for a line are "connected", interpolating the distance. If the pixel buffer has a previous lower distance than the patch's value for that pixel, then there is another patch that is nearer to the source patch for that pixel. Otherwise, the destination patch is nearer, therefore the patch id and distance of that pixel are updated. The similar z-buffer [21] process is exploited in other computer graphics problems such as rendering of the images, in which case the color for that pixel is stored instead of the patch id in the z-buffer.

2.3 Progressive Refinement Radiosity

The conventional radiosity approach has two major disadvantages that limit the usage of the method for complex scenes with large number of patches. First, it requires $O(N^2)$ time to construct the form-factor matrix (Eq. 2.5) (N is the total number of patches), and the environment cannot be viewed until this operation is completed. Second $O(N^2)$ memory is needed to store the coefficient matrix. The *progressive refinement radiosity* [16] approach provides a solution to these two problems by reformulating the conventional radiosity algorithm. This approach requires $O(N)$ time and memory.

The progressive refinement approach differs from the conventional radiosity in two aspects. First, the radiosities of all patches are updated simultaneously. Second, the patches are processed in sorted order according to their energy contribution to the environment.

The method starts with an initial approximation to the light distribution in the scene and approaches to more accurate distribution, providing a graceful and continuous convergence to a realistic looking image.

2.3.1 Simultaneous Update of Patch Radiosities: Shooting vs. Gathering

In the conventional radiosity algorithm, the Gauss-Seidel method is applied to solve the system of equations (Eq. 2.5) for patch radiosities. The method effectively converges to the solution by processing the system of equations one row at a time. The evaluation of the i^{th} row of the matrix provides an estimate of patch i based on the current estimates of the radiosities of all other patches:

$$B_i = E_i + r_i \sum_{j=1}^{N} B_j F_{ij}, \quad 1 \le i \le N \tag{2.14}$$

A single term in Eq. 2.14 gives the light contribution made by patch j to patch i:

$$B_i \; due \; to \; B_j = r_i B_j F_{ij} \tag{2.15}$$

This equation can be reversed by computing the contribution from patch i to patch j using the reciprocity relationship of Eq. 2.3 as follows:

$$\begin{aligned} B_j \; due \; to \; B_i &= r_j B_i F_{ji} && (2.16) \\ &= r_j B_i F_{ij} A_i / A_j && (2.17) \end{aligned}$$

The contribution to any patch j from patch i is computed using Equation 2.17.

Note that, the form-factor row corresponding to patch i is used to distribute the light energy from patch i to all other patches j. Still, the hemicube method can be used for computing the form-factor row. Thus, each step of the progressive refinement radiosity algorithm consists of computing a single row of form-factors for a single patch and adding the light contributions from that patch to all other patches using Eq. 2.17 with computed form-factor values, in effect *shooting* the light from that patch i into the environment.

Figure 2.6 illustrates the difference between conventional (*Gathering*) method versus progressive refinement radiosity (*Shooting*) method.

2.3.2 Solving in Sorted Order

It is desirable for the shooting method to approach to a realistic solution as quickly as possible. At each iteration, the light distribution, that is the radiosity of each patch i, consists of the contributions from the previous shooting patches. The correct distribution is approached faster if the largest contributions are added first. Therefore, at each iteration, the patch with maximum energy should be selected for the method to converge faster.

Different convergence criterias may be used. In the first criteria, the algorithm tests the $\Delta B_i A_i$ of the shooting patch i. If this value is greater than the user-specified tolerance, the algorithm converges. In the second criteria, the $\Delta B_j A_j$ values of all the patches j in the environment are summed, and tested if this area-weighted sum reaches a specified percentage of the initial energy undistributed in the algorithm.

A patch may be selected as the shooting patch more than once during the course of the execution, if it receives more light from the environment. In that case, the environment will already store the estimate of the last shooting from the patch. Hence, a *delta radiosity* ΔB is stored in addition to the radiosity B for each patch in order to store the difference between the previous estimate and the current estimate of the patch radiosity, that is the light the patch has gathered since the last shooting from the patch. When solving in sorted order, the solution tends to proceed in approximately the same order as the light propagates through the environment. Figure 2.7 illustrates the pseudo-code for the progressive refinement radiosity algorithm.

2.3.3 The Ambient Term

The progressive refinement algorithm starts with a dark environment and gradually brightens to a globally illuminated scene. In order to view approximately illuminated environments an ambient term is added [16] during the rendering process. The ambient term allows approximate viewing of the initially dark environments. This term decreases as the algorithm converges to more accurate solutions of B_j's with increasing number of shooting patches. Note that, the ambient term is used for display only and is not used in the solution phase.

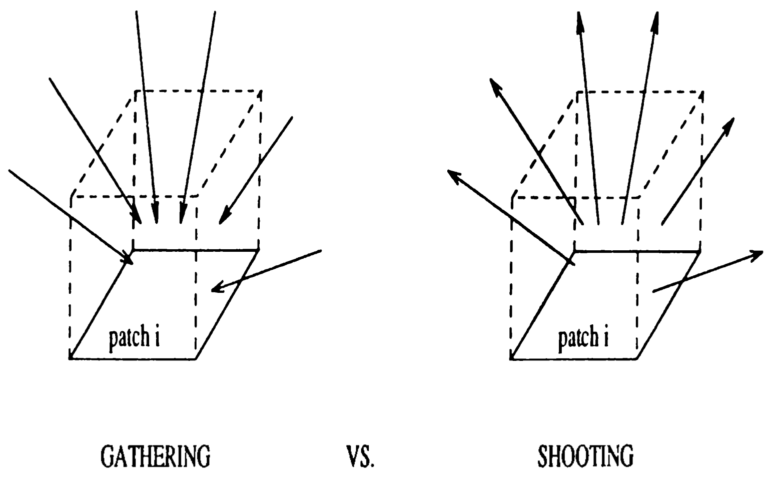


Figure 2.6. Shooting versus Gathering (After Cohen *et al*)

```
/* Initially, $B_i = \Delta B_i = E_i$ for each patch */
/* $E_i = 0$ for non-light sources */
Select patch $i$ with greatest $\Delta B_i A_i$.
while not converged do
    Calculate form factors at patch i (e.g.using Hemicube Algorithm)
    for each patch $j$ do
        $\Delta Rad = r_j \Delta B_i F_{ij} A_i / A_j$;
        $\Delta B_j = \Delta B_j + \Delta Rad$;
        $B_j = B_j + \Delta Rad$;
    endfor
    $\Delta B_i = 0$
    Select next patch $i$ with greatest $\Delta B_i A_i$.
endwhile
```

Figure 2.7. Progressive Radiosity Algorithm

2.4 Further Improvements of the Method

Although the radiosity method has been successful at generating realistic-looking images, it still has computation and modelling requirements that limit the usage of the method. In this section, a brief overview of these requirements and current research for improving the method will be presented. A summary of previous research on radiosity can be listed as follows:

1. Meshing and preprocessing techniques in order to obtain more accurate and faster solution.
2. Form-factor techniques in order to compute more accurate geometrical relationships among the input patches.
3. Adapting the method for dynamic environments.
4. Including more general reflection models and surface properties such as specularity or participating media in the environment.
5. Exploiting parallelism in order to achieve faster image generation speeds without compromising image quality.

1. Meshing and Preprocessing Techniques: The scene models for the radiosity method must satisfy certain constraints in order to obtain an accurate image.

First, *model geometry* requirements state that the input patches have continuous normals and homogeneous material properties (such as reflectivity, emission, etc.), the patch dataset is a solid model (the points are classified as inside, on, outside the object), the facets are single sided with consistent normals, and no two faces overlap each other.

Second, the *meshing* requirements require that no T-vertices occur between neighbouring patches. An example is illustrated in Figure 2.8. Also the polygons must be well-shaped, that is the ratio of the radius of the inscribed circle to the radius of the circumscribed circle should be close to unity, the patches should not be too small or too large, the patches in the environment should be subdivided for better accuracy, and no shadow leakage should occur.

Considering these critical issues, several methods have been proposed in order to speed up the solution by using preprocessing techniques for hierarchical representation of the objects or by starting with coarse patches and adaptively subdividing the necessary regions of the environment during the course of the solution.

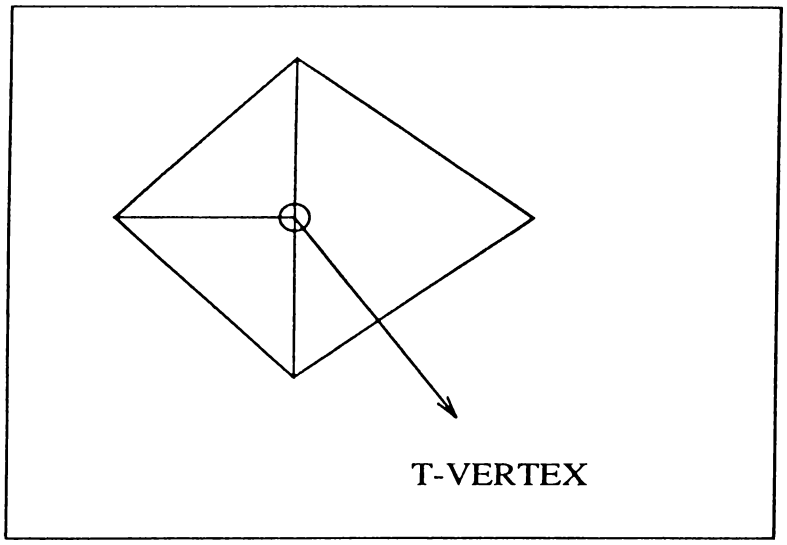


Figure 2.8. T-Vertex

The initial work in adaptive subdivision is by Cohen *et al.* [15]. In this work, they propose a subdivision method called "substructuring" with hierarchically subdivision of the input surfaces into subsurfaces, patches and elements. Baum *et al.* state the ultimate constraints required by the radiosity method and propose automatic subdivision method [5]. Campbell and Fussell [10] present the subdivision for elimination of light leakage.

Hanrahan *et al.* [25] propose a hierarchical representation of the environment. In their algorithm, the hierarchical radiosity is inspired by the N-Body problem, and the patch-to-patch visibility is computed and the form-factor computations are performed with required precision. Smits *et al* [37] propose a view-dependent solution based on Hanrahan's work.

Lischinski *et al.* [29] propose an accurate radiosity based on *discontinuity meshing*. The mesh explicitly represents the discontinuities in the radiance function as boundaries between mesh elements. Piecewise quadratic interpolation is used to approximate the radiance function. This solution is fully automatic and view-independent; and is not limited to constant-radiosity patches and produces less number of patch elements.

Teller and Sequin [40] propose a visibility prepocessing scheme for interactive walkthroughs in constant environments. The method divides the environment into cells and creates a data structure for cell-to-cell visibility. Then, this data structure is used at the walkthrough stage by computing the cell the viewer is in. So, the potentially visible cell is found and only the patches in this cell are used for rendering the frames.

2. Form-Factor Computation Techniques: The hemicube method is an approximation to the hemisphere computation. Although the hemicube method is efficient and it handles occlusions within the environment; it has assumptions which results in incorrect computation of the form-factors.

These assumptions are [3] :

Proximity Assumption The hemicube method assumes that the distance between surfaces i and j is great compared to their size, thus the method samples the patch i with its center point for form-factor computation. This assumption is violated when the two patches are close to each other.

Visibility Assumption The visibility assumption requires that the visibility between differential areas stays constant across patch i, but this assumption can be violated as shown in Figure 2.9(b). In Fig. 2.9(b), the center of surface 1 has a complete view of surface 2, however shaded part of surface 1 is occluded by surface 3. Thus, the hemicube algorithm will overestimate F_{12}.

Aliasing Assumption This assumption states that the surfaces project exactly onto the hemicube face pixels, similar to the aliasing assumption in rendering and ray tracing. As a result of this assumption, the form-factor value for a surface may be overestimated or underestimated as shown in Figure 2.9(c).

In order to solve the problems caused by the hemicube method assumptions and compute more accurate form-factors, a number of form-factor computation techniques have been proposed. Following is a brief summary of these techniques.

The earliest radiosity paper [22] uses direct numerical integration, using Stoke's Theorem for converting the double integral of the form-factor equation

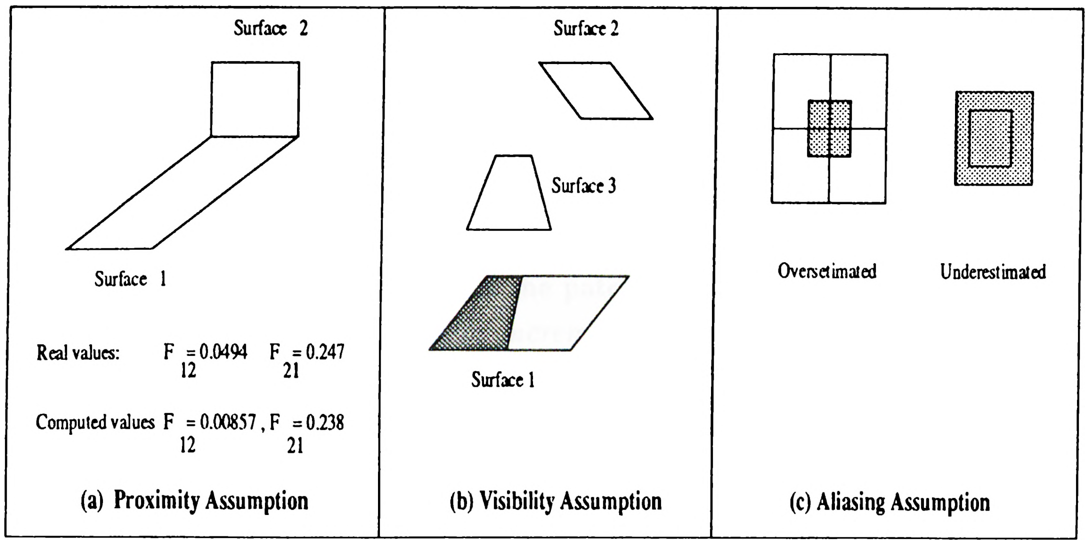

Surface 2
Surface 1
Real values: $F_{12} = 0.0494$ $F_{21} = 0.247$
Computed values $F_{12} = 0.00857$, $F_{21} = 0.238$
(a) Proximity Assumption
Surface 2
Surface 3
Surface 1
(b) Visibility Assumption
Overestimated
Underestimated
(c) Aliasing Assumption

Figure 2.9. Assumptions of the Hemicube Method (After Baum *et al*)

(Eq. 2.8) into a double contour integral. Nishita and Nakamae [30] propose a similar solution, by computing the occlusions among the objects.

In Bu and Deprettere's approach [7], the form-factor vector is computed by locating a hemisphere over the patch and casting rays towards the hemisphere pixels.

Wallace *et al.* propose [43] a source-to-vertex ray tracing solution. Instead of shooting rays from the source patch in the uniform directions of hemicube pixels, their method samples the shooting patch from the point of view of each other surface in the environment. From each vertex in the environment, rays are sampled towards different parts of the shooting patch, and tested for occlusions by other surfaces.

3. Adapting the Method for Dynamic Environments: The construction of the form-factor vector is the most compute-intensive part of the radiosity method. When the geometry of the scene changes, these values change and should be re-computed. Some methods have been proposed for providing efficient solutions for dynamic environments.

Baum *et al.* [2]'s method is used for animation. The method computes an animation sequence and the path of the object movement is used for decreasing the form-factor computation time.

A ray-tracing based solution [8] was proposed by Buckalew *et al.* in order to be used for incremental updates in geometry and color of the patches.

Chen's approach [13] was proposed in order to be used for interactive manipulation of the objects, based on progressive refinement radiosity. The method works by shooting negative light when a light source is turned off, and computing the incremental radiosity of the patch when the light attribute of the patch is modified, and computing *incremental form-factor* for a patch when the geometry of the patch changes.

4. Including More General Reflectance Models: The original radiosity method assumes ideal Lambertian surfaces, that is the surfaces reflect the incoming light with the same amount in all directions. However, specular reflections and transmissions should be considered for more realistic scenes.

Immel and Cohen [27] propose a method which includes specular reflection. In their approach, a relationship between a given outgoing reflection direction for a patch and all outgoing directions for all other patches is constructed, and a simultaneous solution of the resulting system of equations gives an intensity in each direction for each patch.

Wallace and Cohen [42] propose a two-pass solution. The first pass is view-independent and is based on the hemicube algorithm, with extensions to include the effect of diffuse transmission, and specular-to-diffuse reflection and transmission. The second pass is view-dependent based on distributed ray tracing with z-buffer for specular reflection and transmission.

5. Exploiting Parallelism: Although methods that improve the accuracy and speed of the radiosity method have been presented in the literature, still excessive amount of computation is required to simulate the light distribution. Parallelism can be exploited in order to speed up the algorithm further without compromising image quality. In subsequent chapters, the previous work on parallelization of the radiosity method is discussed and two novel approaches for parallelization of progressive refinement radiosity will be presented.

2.5 Conclusion and Summary

In this chapter, the global illumination and progressive refinement radiosity is introduced. To summarize, the progressive refinement radiosity algorithm is an iterative approach. Each iteration consists of the following phases:

1. Shooting Patch Selection.
2. Hemicube Construction for the shooting patch.
3. Conversion of the hemicube item-buffers to a form-factor vector.
4. Contribution computation from the shooting patch to the environment.

In Phase 1, the patch with maximum $\Delta B_i A_i$ is selected as the shooting patch for faster convergence. In Phase 2, all the other patches are projected onto five hemicube faces located on the shooting patch. For this process, the patches are passed through a projection pipeline, and the patches which are not visible leave the pipeline in an early stage. In Phase 3, the hemicube is scanned and delta form-factors of the pixels corresponding to the same patch are added in order to obtain the form-factor value for that patch. Phase 4 consists of evaluating Eq. 2.17 for each patch j, using the form-factor vector that has been computed for the shooting patch.

Chapter 3

Overview of Parallelism in Radiosity

This chapter presents the background for the parallel processing and summarizes the previous efforts for parallelization of the progressive refinement algorithm.

3.1 Classification of Parallel Architectures

In general, parallel architectures can be classified according to:

- the number of instruction and data streams supported,
- the memory organization,
- the coupling,
- the granularity.

Classification According to the Multiplicity of Data and Instruction Streams

This classification follows Flynn's taxonomy [20], and divides the parallel architectures into four classes:

- **SISD** (Single Instruction Single Data) : This is the standard sequential computer.

- **MISD** (Multiple Instruction Single Data) : There is no available parallel computer belonging to this class at this time.
- **SIMD** (Single Instruction Multiple Data) : The processor set executes a single stream of instructions, but each processor operates on its own data. Hence, a single instruction is executed at a time by all the processors.
- **MIMD** (Multiple Instruction Multiple Data) : In this class, each processor in the set executes a different program with different data at a time.

Classification According to Memory Organization

There are two main classes based on this classification:

- **Shared Memory** : In this class, there is a global memory address space accessible by all processors. Processors may also have local memories (caches) that are orders of magnitude smaller and faster compared to global memory. Synchronization is achieved by shared variables.
- **Distributed Memory** : There is no global memory or memory address space in this class. Processors have only local memory; and synchronization and coordination among the processors are achieved through message passing.

Classification According to Coupling

There are two basic classes of parallel computers and algorithms according to this type of classification:

- **Synchronous Architectures** : The processors perform their tasks or communication in lock-step or highly synchronous manner.
- **Asynchronous Architectures** : Tasks or communication are not performed in lock-step fashion. Overhead for data exchange is typically higher, and objects may proceed at somewhat different speeds. Occasionally, barrier synchronization may be used to allow slow objects to catch up with the faster objects.

Classification According to Granularity

Granularity is a way of expressing the ratio of computation to communication in parallel machines. There are three main types of granularity:

- **Coarse-grain Architectures :** Computation to communication ratio is very high, and a few but powerful (may be heterogeneous) processors are exploited.
- **Medium-grain Architectures :** Parallel computers in this class typically have computation to communication ratio of 100 or more.
- **Fine-grain Architectures :** These computers are characterized by a large number of very simple processors. The computation to communication ratio is almost unity. Generally, SIMD type of architectures exploit fine-grain parallelism.

3.2 Design Criteria for Parallelization

The most promising parallel architecture is distributed-memory message-passing architecture which is usually referred as multicomputers. Multicomputers are asynchronous and MIMD type architectures. Multicomputers have nice scalability feature due to the lack of shared resources. In this work, parallelization of progressive refinement radiosity on multicomputers is investigated.

Exploiting good speed-up through parallelism on multicomputers is not straightforward. The parallel program designer must decide on atomic tasks depending on the type of parallelism and granularity selected. In order to obtain good performance, the parallel graphics algorithms must be developed considering the following issues:

1. The decision of decomposition of the tasks to processors,
2. Balancing of the tasks assigned to processors,
3. Selection of the granularity of the tasks for processors so as to match the target architecture and the application,
4. Exploiting the graphical coherence,

5. Careful distribution of the data to the processors,
6. Scalability of the parallel algorithm on larger machines.

3.2.1 Type of parallelism

There are two main types of algorithm decomposition: data parallelism and functional parallelism. In *data parallelism*, the data that has to be processed is divided among the processors and each processor performs the operations on its own data. Generally, data parallelism is suitable for SIMD architectures. *Functional parallelism* refers to allocating the processors such that each processor is given a specific task. *Operational parallelism* and *procedural parallelism* are two types of functional parallelism. In operational parallelism, the basic instructions (such as assignment) are selected as concurrent operations. In procedural parallelism, the algorithm is decomposed into coarse sections allocated to different processors. *Pipelining* is a combination of functional and data parallelism. The type of parallelism should be decided considering the target architecture and the problem.

3.2.2 Load Balancing

Load balance is the degree to which work is evenly distributed among the processors. If all the processors have almost equal amount of work to be performed, the parallel program executes to completion more quickly. Hence, the computational load should be evenly distributed among the processors. As the load balance increases, the processor idle time decreases hence increasing the overall processor utilization.

The load distribution can be achieved in two ways: static task assignment and dynamic task assignment. In *static* assignment, the tasks are distributed to the processors prior to the execution of the parallel program. This distribution requires careful examination of the computational tasks and the data in order to obtain even distributions. This approach minimizes the scheduling overhead. In *dynamic* task assignment, computational tasks are allocated to processors during runtime. This type of task assignment can be used if the number of tasks is much larger than the number of processors. The advantages of dynamic task assignment are that scheduling is performed at runtime and the load balance is achieved dynamically and more precisely. However, dynamic task assignment may have large communication overhead due to the re-scheduling operations.

3.2.3 Granularity

Granularity of a parallel program is a way of expressing the ratio of the computation to communication overhead. Granularity of the parallel program must be decided considering the problem and the target architecture.

3.2.4 Exploiting Graphical Coherence

Coherence refers to the usage of the previous calculations for subsequent computations in order to eliminate redundant computations and increase the performance of the algorithms. Coherence is an important typical issue for many computer graphics problems. For example, in scan-conversion of the polygons in the z-buffer hidden-surface elimination, the color and distance values of the pixels for a polygon are evaluated incrementally based on the data of the previous neighbour pixels, similarly in animation, subsequent frames are generated and rendered using the previous frames.

Exploiting parallelism requires partitioning of the computations among the processors. This may cause previous computations that would be used for the sequential program are not used in the parallel algorithm because another processor may compute the previous values concurrently, thus losing the coherence. Hence, more computations would be required for computing previous tasks. The solution of this conflict requires carefully decomposing parallel algorithms in order to exploit utmost coherence.

3.2.5 Data Access

Typically, graphics algorithms deal with large amounts of data and management of these data is one of the important issues in parallel algorithm development. For example, in order to render an image with satisfactory quality, at least 1 to 2 Megabytes of main memory is required. Also, for the radiosity method, a scene consisting of 10^5 patches is not impossible, and this data require more than 5 Megabytes of memory for storing only the patch dataset. So, it is not practical to replicate the whole dataset in each processor in a multicomputer. This problem becomes more crucial for massively parallel computers. The data decomposition must be carefully performed considering the problem decomposition and other facts such as load balancing. Remote data access takes considerably more time than local access, therefore should be minimized in order to obtain good performance.

3.2.6 Scalability

Designing scalable parallel algorithms becomes more important as the technology advances. The parallel algorithm must be designed so that it can execute on larger number of processors without decreasing the efficiency of the solution significantly.

3.3 Parallelism in Radiosity and Previous Work

The parallel radiosity algorithms can be classified according to the level of parallelism they exploit. In this section, the possible approaches to radiosity are discussed and the parallel solutions proposed in the literature are presented.

The levels of parallelism in the progressive refinement radiosity method can be classified in two main levels:

1. **Level 1:** More than one form factor rows are computed and more than one patch shoot their energy in parallel. Typically, each processor is given a distinct shooting patch and it performs a single iteration for its patch. Two distinct approaches exploiting this level of parallelism are:

   (a) One of the processors serves as the master and the other processors are the slaves. Only the master processor maintains the patch radiosities, and the slave processors are given their shooting patches by the master on a demand-driven basis. The slaves compute the form-factor vector for their shooting patch.

   (b) The shooting patch selection for the processors is done in a distributed manner. In this type of parallelization, there is no master processor and global interprocessor communication is performed for form-factor computation and light energy distribution among the processors.

2. **Level 2:** The processors work together for shooting from a single patch to the environment simultaneously, thus only one patch shoots its energy at a time.

   The algorithms that belong to this level of parallelism typically exploit data parallelism, and can vary from fine-grain to medium-grain parallelism. Either *image space decomposition* or *object space decomposition* can be used. In image space decomposition, the hemisphere (hemicube) corresponding to the shooting patch is divided into parts which are assigned to different groups of processors. In *object space decomposition* each processor group processes a subset of the patch data on the replicated hemicube.

Algorithms that exploit both of these levels have been proposed in the literature. The first level of parallelism deviates from the sequential algorithm so that the sequence of shooting patches selected by the parallel algorithm differs slightly from that of the sequential algorithm. This issue will be investigated in detail in the subsequent chapters. Level 1 has been investigated more than level 2 in the literature.

3.3.1 Parallelization: More than One Patch at a Time

In this level of parallelism, typically medium-to-coarse grain parallelism is exploited. The previous work belonging to this level is as follows:

a. Master-Slave Shooting Patch Selection

Typically, this approach for parallelism is suitable for coarse-grain parallelism. Different types of architectures such as network of workstations and transputers have been used.

Recker, George, Greenberg

Recker *et al.* [34] use coarse-grain loosely-coupled HP385 workstations, linked by Ethernet. The 3D geometry data of the patches is duplicated on the slaves and only the master maintains patch radiosities. The master processor is directly connected to each slave processor. The shooting patch is assigned to the slave processors by the master processor on a demand-driven basis. The slave processors compute the form-factor vector corresponding to their own shooting patch, and send this vector to the master processor, and the master updates the radiosity and delta radiosity values using this form-factor vector.

As the experimental results they obtain indicate, the master processor becomes a bottleneck with increasing number of processors and the efficiency decreases to 0.4 for 12 slave processors.

Feda, Purgathofer

Feda and Purgathofer use INMOS T800 Transputer Network, arranged in two groups: worker network and renderer network. The 3D geometry of the scene is distributed among the processors. The master processor stores the best candidates for shooting patches selected among the maximums of the local patches of the processors. A slave processor requests for a shooting patch from the master processor by sending its local with maximum energy, and the master processor sends that processor the best candidate it holds in its own memory. Then, the master updates its sorted list of shooting patch candidates using the received patch data.The slaves are connected in a minimum path length network. The local patch data of the processors are circulated globally in blocks around the network so that all patches visit all the processors. The slave processors compute the form-factor vectors and then the update vectors corresponding to their local shooting patches. These update vectors are routed globally in the network so that all slave processors update the radiosity and delta radiosity values for their local patches.

In their paper [19], Feda and Purgathofer analyze the performance of their solution experimentally under different conditions. They observe that increasing the number of simultaneously processed shooting patches at one slave processor increases the performance of the parallel solution, because the computation to communication ratio (granularity) increases with more than one shooting patches per processor. However, increasing the number of simultaneously processed hemicubes puts a limit to the complexity of the input scene because of the memory limitations.

b. Distributed Shooting Patch Selection

Chalmers, Paddon

Chalmers and Paddon [12] examine the implementation of progressive refinement radiosity on T800 Transputers arranged in different configurations such as ring, hypercube, torus, AMP. They observe that the average distance between processors in the network affects the performance of the parallel solution.

In their work, the 3D geometry data is distributed among the processors, and each processor selects its shooting patch as the patch with maximum energy among its local patches. The processors "fetch" the local patch geometry data of other processors and project the patches onto their local hemicube. The patch data are globally circulated in an asynchronous manner. When a processor completes the form-factor computation for its shooting patch, the form-factor vector is "shot" to other processors. Each processor, upon receiving a form-factor vector, computes the contributions from that shooting patch to its local patches using the corresponding entries.

Chalmers and Paddon present their experimental results and conclude that the performance of their parallel solution is maximized with AMP configuration as this configuration decreases the message density. They observe that in the ring and torus configurations, message saturation occurs with increasing number of processors.

Bouatouch, Menard, Priol

Bouatouch *et al.* present parallel radiosity algorithm using shared virtual memory. They use the KOAN Shared Virtual Memory embedded onto the operating system of Intel's iPSC/2 [6].

In their work, the form-factor vectors for the shooting patches are computed using ray-casting, instead of the hemicube method with item-buffers. Instead of dividing the surfaces into patches and storing the patches explicitly, they store the geometry data for surfaces only and find the intersected patch by a given ray by computing the local parameters inside the surface using the *patch texture* data structure. The scene geometry as well as the radiosity and delta radiosity values for the patches are shared by the processors, while each processor holds a distinct form-factor vector in its own local memory.

For shared memory architecture, selection of the shooting patch and contribution computation must be performed in critical sections in order to obtain consistency. However, the critical section is not scalable to large number of processors. To increase the performance of their solution, they propose a scheme in which the update of the patch radiosities and selection of the next shooting patch are performed at the same time in the same critical section.

Baum, Wignet

Baum and Wignet [4] experiment parallelization of the method on a shared memory multiprocessor architecture (Silicon Graphics Iris 4D 280). The algorithm is partitioned to the producer which performs the hemicube production phase of the progressive radiosity algorithm by a special hardware for z-buffer, and consumer processes which perform the other steps such as the patch selection, contribution computation, implemented in software. Their aim is to have the special hardware do the most time-consuming part of the algorithm, that is the hemicube production phase.

Guitton, Roman, Schlick

Guitton *et al.* [24] propose two parallel approaches for progressive refinement radiosity based on ray tracing, stochastic and deterministic approaches. For the shooting process from a tile, rays are cast randomly in the stochastic approach, and towards each potentially intersecting patch in the deterministic approach. The 3D environment is divided into slices, each slice corresponding to a processor, and rays are communicated among the slices connected in a bidirectional ring topology.

3.3.2 Parallelization: One Patch at a Time

In this type of parallelism, only one form-factor vector corresponding to a single shooting patch is computed at a time. The parallelism in this level typically exploit data parallelism. The algorithms belonging to this level vary from fine-level to medium-level parallelism.

Varshney, Prins

Varshney and Prins [41] propose an environment projection approach for a *single-plane* form-factor computation method which was introduced by Recker *et al* [34]. The single plane is an approximation to the hemicube in order to obtain fast form-factor computation. The single-plane approximation is shown in Figure 3.1.

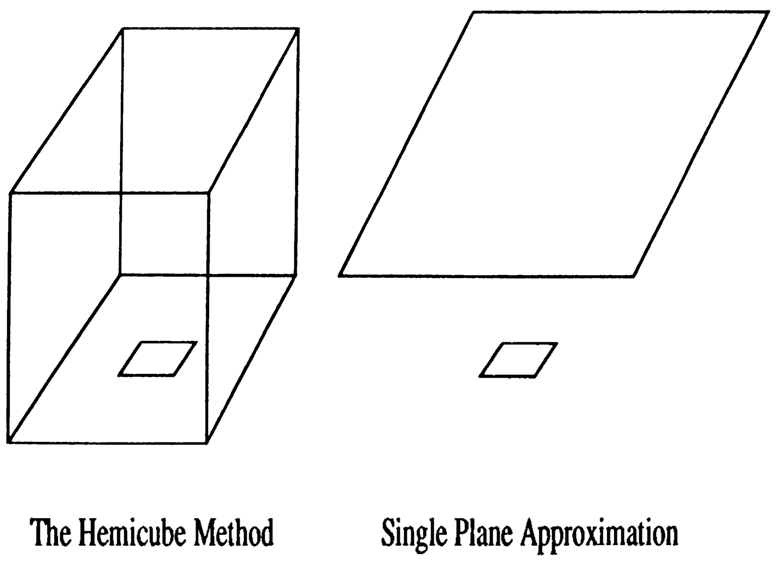


Figure 3.1. Single Plane Approximation (After Recker *et al.*)

They investigate the parallelization of the progressive refinement approach on fine-grain, mesh-connected SIMD MasPar MP-1. The patch data is distributed evenly among the processors. The single-plane pixels are partitioned among the 2D processor mesh so as to achieve an orthogonal and monotonic correspondence between single plane x and y coordinates and the processor indices.

The algorithm proceeds as follows. Having selected the shooting patch, all processors compute the projection coordinates of their local patch vertices and find the bounding boxes in the image space. Then, processors send their local patch geometry data to the processors corresponding to the upper-left corners of the bounding boxes using the global routing network of the MasPar MP-1. Next, in the scan-conversion phase, the processors spread the projected patch data they receive to the right and downwards using the local X-net communication of the MasPar MP-1, so that each processor collects the patch data that project onto its local pixels. Then, the problem reduces to finding the patch with minimum depth in the local pixels. During the contribution phase, the processors send the delta form-factor values corresponding to their local pixels to the owners of the patches projected onto their local pixels, and the receiving processors compute the delta radiosity and radiosity values for their local patches.

Their algorithms have the advantage that memory is efficiently used and coherence in scan-conversion phase can be exploited. However, the performance of the algorithm may degrade if the patches are not distributed equally on the single-plane image, increasing the load imbalance. Furthermore, the form-factors and the contributions are not computed accurately because of the limitations of the single-plane approximation.

Drucker, Schröder

Drucker and Schröder propose a parallel progressive refinement algorithm on CM-2 multiprocessor [18]. They use the ray-tracing algorithm proposed by Wallace *et al.* [43]. Recall that, in Wallace's ray-tracing algorithm; for each destination vertex in the environment, rays are sampled towards the shooting patch, and tested for intersection with the shooting patch in order to compute the form-factor vector from the shooting patch.

Drucker and Schröder apply a *processor allocation* technique in order to solve the visibility problem with the shooted rays. In this technique, the number of processors required to perform a task is computed initially, and then these processors are allocated to the requesting processor to perform the required task.

In their approach, initially, the world space is first discretized into constant sized voxels, each voxel containing the objects that intersect it. The algorithm proceeds as follows. Each processor generates the sampling rays from the vertices it contains in its local voxels. As each generated ray passes through a different number of pixels, each *ray-processor* computes the number of required voxels and its ray will pass, and that number of *voxel-processors* are allocated in turn. So, every processor contains a voxel-ray pair. Then, each voxel-processor finds the number of objects contained in it and allocates that number of *object-processors*. Then, each processor can compute the intersection of a single ray with a single object. Then, the intersection tests of all rays with all the objects can be executed simultaneously.

Their algorithm takes time proportional to the number of objects times the number of rays generated, hence performs well for a large number of objects with few rays or vice versa. However, the disadvantages of their solution is that unnecessary work is performed as all the voxels are tested for intersection, hence coherence is lost in intersection computations.

3.4 Critical Issues of the iPSC/2 Hypercube

As is discussed in the Section 3.3, the progressive refinement method has been investigated for parallelization on many parallel architecture platforms. In this thesis, Intel's iPSC/2 multicomputer is used as the target architecture. In this section, the specifications of the iPSC/2 and the basic global parallel operations are presented.

The Intel Personal Supercomputer iPSC/2 is an MIMD type multicomputer. The processors (or nodes) are connected in a d-dimensional hypercube topology with $P = 2^d$ nodes labeled from 0 to $2^d - 1$. Each node is a (≈ 4 MIPS) Intel 80386 based processor with a 80387 math floating point coprocessor and 4 megabytes of memory. The nodes are controlled by a front-end processor (the host), and all communications are performed by message passing communication system.

The communication between the processors in the iPSC/2 system is done by high speed message passing using the Direct Connect Module facility. The system libraries provide a variety of communication primitives such as SEND, RECV, GSUM (Global SUM), GCOL (Global COLLECT).

In the iPSC/2 system, there are two bit-serial and full duplex links that interconnect nearest neighbour processors. This configuration allows simultaneous bidirectional message traffic between nearest neighbour processors. The communication protocols for $GVSUM$ and $GCOL$ functions are very similar in nature. These communication protocols exploit the two bidirectional links to overlap nearest neighbour communications. Only the parallel algorithm for

GSUM (Global vector SUM) is given below. Extension of this algorithm to the *GCOL* operation is straightforward.

Let each processor of a d-dimensional hypercube be represented by a d-bit binary number $(b_{d-1}...b_0)$ stored in its local memory variable *mynode*. Also define channel-i as the set of (2^{d-1}) bidirectional communication links connecting two neighbour processors whose binary representations differ only in bit position-i. The steps of the *GSUM* algorithm is illustrated in Figure 3.2. At the end of d concurrent exchanges and summations, each processor holds the global sum at its local memory in variable *sum*. Figure 3.3 illustrates the communication protocol with the channels used at each iteration.

```
for i=0 to d-1 do
    dnode = mynode $\oplus$ $2^i$;
    send sum [1..N] to dnode via channel-i;
    receive temp [1..N] from dnode via channel-i;
    for j=1 to $N$ do
        sum[j] = sum[j] + temp[j];
    endfor
endfor
```

Figure 3.2. Algorithm for Parallel GSUM Operation

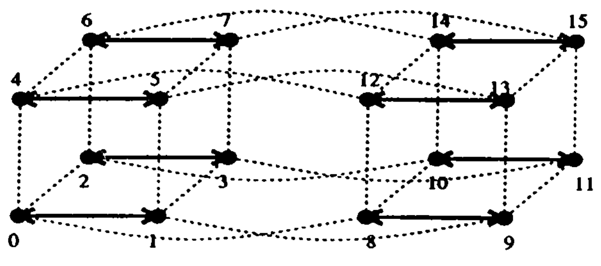
The parallel complexity of the communication protocol used in the *GSUM* operation is:

$$T_{GSUM} = d \times t_{SU} + d \times N \times t_{TR} + d \times N \times T_{ADD} \qquad (3.1)$$

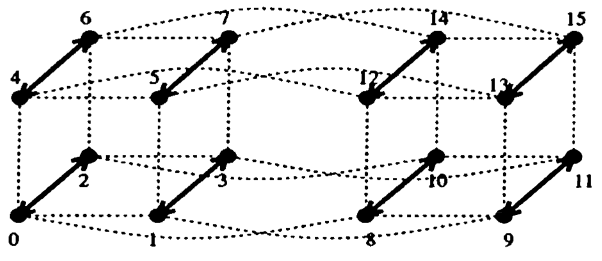
where N is the size of the local initial and the resultant vectors. Here, t_{SU} denotes the set-up time (zero byte latency time) and t_{TR} denotes the transmission time per floating point word. In a coarse grain multicomputer, $t_{SU} \gg t_{TR}$. For example, in iPSC/2, $t_{SU} = 550\mu seqs$ and $t_{TR} = 1.44\mu seqs$ per 4-byte word.

3.4.1 Embedding the ring onto hypercube

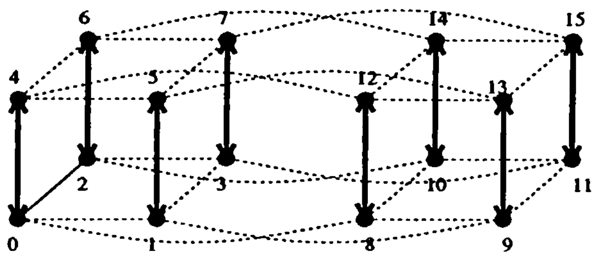
Any ring with an even number of processors can be embedded onto the hypercube topology, so that there is a direct interconnection between the neighbour processors in the ring (chain). If the number of processors in the ring is a power of 2, then the ring can be embedded onto the hypercube topology using the Gray Code ordering scheme. Example embeddings of rings with different number of processors are illustrated in Figure 3.4.



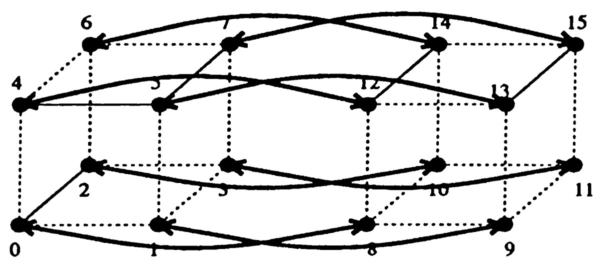
COMMUNICATION VIA CHANNEL 0



COMMUNICATION VIA CHANNEL 1



COMMUNICATION VIA CHANNEL 2



COMMUNICATION VIA CHANNEL 3

Figure 3.3. Communication Protocol for global operations on the hypercube

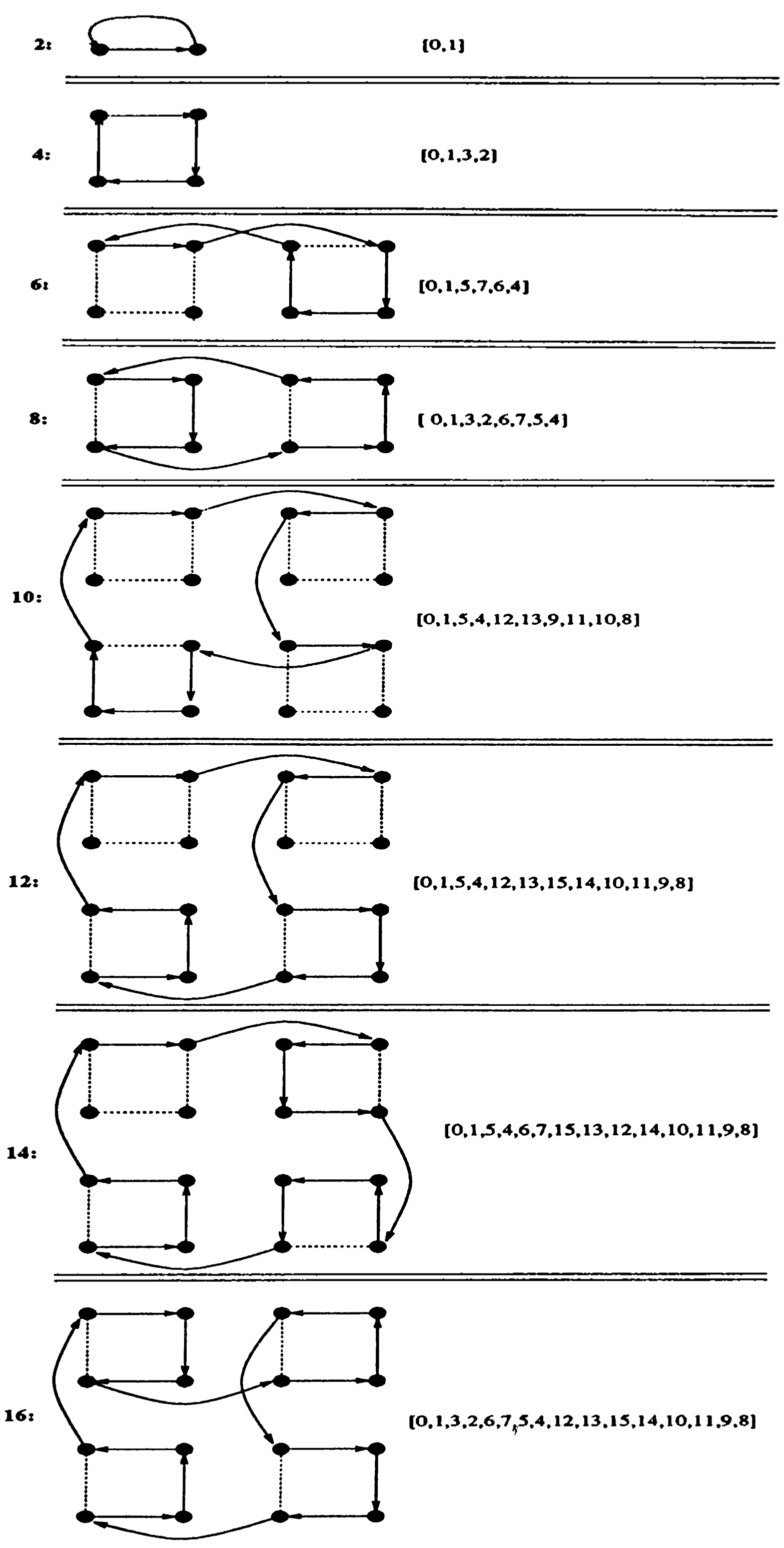


Figure 3.4. Ring embedding onto the hypercube

Chapter 4

Parallelization: Patch Data Circulation

This chapter presents the first proposed parallelization scheme which is based on processing more than one shooting patch at a time. This scheme exploits the Level 1b parallelism mentioned in Section 3.3. A synchronous scheme, based on static task assignment, is proposed, in order to achieve better coherence during the parallel light distribution computations.An efficient global circulation scheme is proposed for the parallel light distribution computations, which reduces the total volume of concurrent communication by an asymptotical factor. The proposed parallel algorithm is implemented on iPSC/2 hypercube multicomputer. Load balance quality of the proposed static assignment schemes are evaluated experimentally. The effect of coherence in the parallel light distribution computations on the shooting patch selection sequence is also investigated.

4.1 Introduction

Recall that the progressive refinement radiosity gives an initial approximation to the illumination of the environment and approaches to the correct light distribution iteratively. Each iteration can be considered as a four phase process:

1. Shooting patch selection,
2. Production of hemicube item-buffers,

3. Conversion of item-buffers to a form-factor vector,
4. Light distribution using the form-factor vector.

In the first phase, the patch with maximum energy is selected for faster convergence. In the second phase, a hemicube [14] is placed onto this patch and all other patches are projected onto the item-buffers of the hemicube using the z-buffer algorithm for hidden patch removal. The patches are passed through a projection pipeline consisting of: visibility test, clipping, perspective projection and scan-conversion. In the third phase, the form-factor vector corresponding to the selected shooting patch is constructed from the hemicube item-buffers by scanning the hemicube and adding the delta form-factors of the pixels that belong to the same patch.

In the last phase, light energy of the shooting patch is distributed to the environment, by adding the light contributions from the shooting patch to the other patches. Distribution of light energy necessitates the use of the form-factor vector computed in Phase 3. The contribution from the shooting patch i to patch j is given by [16]:

$$\begin{aligned} \Delta R(r,g,b) &= r_j(r,g,b)\Delta B_i(r,g,b)F_{ij}A_i/A_j && (4.1)\\ B_j(r,g,b) &= B_j(r,g,b)+\Delta R(r,g,b) && (4.2)\\ \Delta B_j(r,g,b) &= \Delta B_j(r,g,b)+\Delta R(r,g,b) && (4.3) \end{aligned}$$

In Eq. 4.1, $\Delta B_i(r,g,b)$ denotes the delta radiosity of patch i, $r_j(r,g,b)$ is the reflectivity value of the patch j for 3 color-bands, A_j denotes the area of the patch j, F_{ij} denotes the j^{th} element of the form-factor vector constructed in Phase 3 for the shooting patch i. During the execution of the algorithm, a patch may be selected as the shooting patch more than once, therefore a delta radiosity value (ΔB) is stored in addition to the radiosity (B) of the patch, which gives the difference between the current energy and the last estimate distributed from the patch (i.e. the amount of light the patch has gathered since the last shooting from the patch). This iterative process is halted when $\Delta B_i A_i$ values for all the patches reduce below a user-specified tolerance value.

4.2 Parallelization

As is mentioned earlier, progressive refinement radiosity is an iterative algorithm. Hence, computations involved in an individual iteration should be investigated for parallelization while considering a proper interface between successive iterations. In this algorithm, strong computational and data dependencies exist between successive phases such that each phase requires the computational results of the previous phase in an iteration. Hence, parallelism at each phase should be investigated individually while considering the dependencies between successive phases. Furthermore, strong computational and data dependencies also exist within each computational phase. These *intra-phase* dependencies necessitate global interaction which may result in global interprocessor communication at each phase on a distributed-memory architecture. Considering the crucial granularity issue in parallel algorithm development for coarse-grain multicomputers we have investigated a parallelization scheme which slightly modifies the original sequential algorithm. In the modified algorithm, instead of choosing a single patch, P shooting patches are selected at a time on a multicomputer with P processors. The modified algorithm is still an iterative algorithm where each iteration involves the following phases:

1. Selection of P shooting patches,
2. Production of P hemicube item-buffers,
3. Conversion of P hemicubes to P form-factor vectors,
4. Distribution of light energy from P shooting patches using these P form-factor vectors.

Note that, the structure of the modified algorithm is very similar to that of the original algorithm. However, the computations involved in P successive iterations of the original algorithm are performed simultaneously in a single iteration of the modified algorithm. This modification increases the granularity of the computational phases since the amount of computation involved in each phase is duplicated P times. Furthermore, it simplifies the parallelization since production of P hemicube buffers (Phase 2) and production of P form-factor vectors (Phase 3) can be performed simultaneously and independently. Hence, processors can concurrently construct P form-factor vectors corresponding to P different shooting patches without any communication.

The modified algorithm is an approximation to the original progressive refinement method. The coherence of the shooting patch selection sequence is disturbed in the modified algorithm. The selection of P shooting patches at a time ignores the effect of the mutual light distributions between these patches and the light distributions of these patches onto other patches during this selection. Thus, the sequence of shooting patches selected in the modified algorithm may deviate from the sequence to be selected in the original algorithm. This deviation may result in a greater number of shooting patch selections for convergence. Hence, the modification introduced for the sake of parallelization may degrade the performance of the original algorithm. This performance degradation is likely to increase with the increasing number of processors. This chapter presents an experimental investigation of this issue.

In Chapter 3, various parallel progressive radiosity algorithms that were proposed in the literature have been discussed. The algorithmic modification presented in this chapter is similar to the parallel implementations classified in Level 1.(b). However, these parallel implementations utilize an *asynchronous* scheme. These asynchronous schemes have the advantage of minimizing the processors' idle time since form-factor and light distribution computations proceed concurrently in an asynchronous manner. However in these schemes a processor, upon completing a form-factor vector computation for a shooting patch, selects a new shooting patch for a new form-factor computation. Hence, this shooting patch selection by an individual processor does not consider the light contributions of the form-factor computations concurrently performed by other processors. Furthermore, asynchronous communication among the processors can create congestion on the communication network, especially in simple topologies such as ring [12]. In this work, we propose a synchronous scheme which is expected to achieve better coherence in the distributed shooting patch selections and eliminate message saturations. The parallelization of the proposed scheme is discussed in the following sections.

4.2.1 Phase 1: Shooting Patch Selection

There are two alternative schemes for performing this phase: local shooting patch selection and global shooting patch selection. In the local selection scheme, each processor selects the patch with maximum $\Delta B_i A_i$ value among its local patches. In the global selection scheme, each processor selects the first P patches with the greatest $\Delta B_i A_i$ value among its local patches in a sorted order and interprocessor communication is performed in order to obtain the first P patches with maximum energy among these patches. Then, each processor selects a distinct shooting patch among these maximal patches.

The number of shooting patch selections required for convergence of the parallel algorithm to the user-specified tolerance depends on the shooting patch selection scheme. Global scheme is expected to converge more quickly because the patches with *globally* maximum energy are selected. However, in the local scheme, the shooting patches that are selected may deviate largely, if maximum energy holding patches are gathered in some of the processors, while the other processors hold less energy holding patches. Hence, the global scheme is expected to achieve better coherence in distributed shooting patch selection. However, the global scheme requires circulation and comparison of P buffers, hence necessitating global communication overhead. Therefore, the global communication scheme should be designed efficiently considering the interconnection topology of the processors in order to minimize this overhead. In this work, we present efficient communication schemes for global patch selection for the ring and hypercube topology.

Ring Topology

First, each processor selects the first P patches with the greatest $\Delta B_i A_i$ value among its local patches in sorted order and puts these patches (together with their geometry and color data) into a local buffer in decreasing order according to their $\Delta B_i A_i$ values. Then, these buffers of sizes P are circulated in $P-1$ concurrent communication steps as follows. In each concurrent step, each processor merges its sorted buffer of size P with the sorted buffer received of size P, discarding P patches with smaller $\Delta B_i A_i$ values. Then, each processor sends the resulting buffer to the next processor in the ring. Note that, each processor keeps its original local buffer intact during the circulation. At the end of $P-1$ communication steps, each processor holds a copy of the same sequence of P patches with maximum $\Delta B_i A_i$ values in decreasing order. Then, processor k selects the k^{th} patch in the local sorted patch list, for k=0,1...$P-1$.

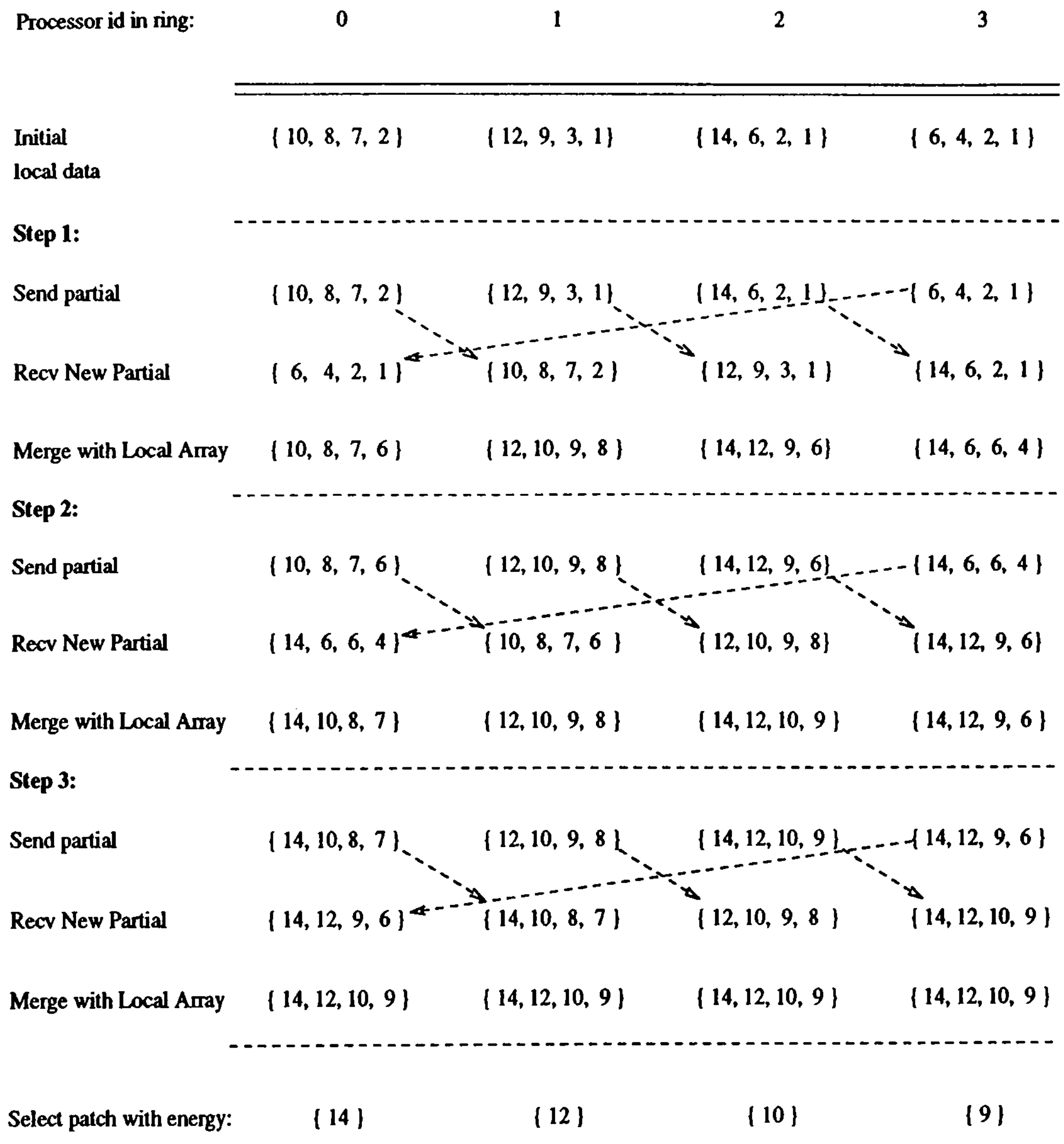
The parallel algorithm for shooting patch selection for the node processors in the ring topology is given in Figure 4.1 and a sample execution for a ring of 4 processors is illustrated in Figure 4.2.

```
/* LocalMaximums :  Holds P patches with max radiosity locally;
   PartialMaximums: Holds partial maximums during circulation;
   Entries of LocalMaximums are not modified during circulation.  */

nextnode = ( mynode + 1 ) mod P;

1.  Select the P local patches with maximum $\Delta B_i A_i$
   into array LocalMaximums
2.  Select the P patches among the $P^2$ maximums.
   2.1.  for i=0 to P-1 do
              PartialMaximums[i] = LocalMaximums[i];
   2.2.  for i=1 to P-1 do
              send PartialMaximums to nextnode;
              receive into PartialMaximums;
              Find the P maximums among entries of PartialMaximums
                 and LocalMaximums and store these values
                 into PartialMaximums;
         endfor;
3.  Select the processor's local shooting patch from the result.
   ShootingPatch = PartialMaximums[mynode];
```

Figure 4.1. Algorithm for global shooting patch selection on ring topology



Processor id in ring:	0	1	2	3
Initial local data	{ 10, 8, 7, 2 }	{ 12, 9, 3, 1 }	{ 14, 6, 2, 1 }	{ 6, 4, 2, 1 }
Step 1:				
Send partial	{ 10, 8, 7, 2 }	{ 12, 9, 3, 1 }	{ 14, 6, 2, 1 }	{ 6, 4, 2, 1 }
Recv New Partial	{ 6, 4, 2, 1 }	{ 10, 8, 7, 2 }	{ 12, 9, 3, 1 }	{ 14, 6, 2, 1 }
Merge with Local Array	{ 10, 8, 7, 6 }	{ 12, 10, 9, 8 }	{ 14, 12, 9, 6 }	{ 14, 6, 6, 4 }
Step 2:				
Send partial	{ 10, 8, 7, 6 }	{ 12, 10, 9, 8 }	{ 14, 12, 9, 6 }	{ 14, 6, 6, 4 }
Recv New Partial	{ 14, 6, 6, 4 }	{ 10, 8, 7, 6 }	{ 12, 10, 9, 8 }	{ 14, 12, 9, 6 }
Merge with Local Array	{ 14, 10, 8, 7 }	{ 12, 10, 9, 8 }	{ 14, 12, 10, 9 }	{ 14, 12, 9, 6 }
Step 3:				
Send partial	{ 14, 10, 8, 7 }	{ 12, 10, 9, 8 }	{ 14, 12, 10, 9 }	{ 14, 12, 9, 6 }
Recv New Partial	{ 14, 12, 9, 6 }	{ 14, 10, 8, 7 }	{ 12, 10, 9, 8 }	{ 14, 12, 10, 9 }
Merge with Local Array	{ 14, 12, 10, 9 }	{ 14, 12, 10, 9 }	{ 14, 12, 10, 9 }	{ 14, 12, 10, 9 }
Select patch with energy:	{ 14 }	{ 12 }	{ 10 }	{ 9 }

Figure 4.2. Example global shooting patch selection on ring topology

Hypercube Topology

The parallel algorithm for the hypercube topology is slightly different from the algorithm for the ring topology. The hypercube algorithm for global shooting patch selection uses the communication protocol illustrated in Figure 3.3 and performs the operation in $d = log_2 P$ steps as follows: In each concurrent step, each processor exchanges its partial result with another processor along the channel for that iteration; and merges this received partial result with the previous data the processor has sent in this iteration, discarding the P patches with lowest $\Delta B_i A_i$.

The parallel algorithm for the node of the SIMD hypercube is shown in Figure 4.3 and an example execution for 4 processors is shown in Figure 4.4.

```
/* LocalMaximums :  Holds P patches with max radiosity locally;
   PartialMaximums: Holds partial maximums during circulation;

1.  Select the P local patches with maximum $\Delta B_i A_i$
    into array LocalMaximums
2.  Select the P patches among the $P^2$ maximums.
    2.1.  for i=0 to P-1 do
              PartialMaximums[i] = LocalMaximums[i];
    2.2.  for i=0 to (d-1) do
              dnode = mynode $\oplus$ $2^i$;
              send PartialMaximums to dnode;
              receive ReceivedMaximums from dnode;
              Find the $P$ maximums of entries of PartialMaximums and
                 ReceivedMaximums and store these values
                 into PartialMaximums;
          endfor;
3.  Select the processor's local shooting patch from the result.
    ShootingPatch = PartialMaximums[mynode];
```

Figure 4.3. Algorithm for global shooting patch selection on hypercube topology

Performance Analysis of Phase 1

The parallel complexity of the communication phase for shooting patch selection on the ring topology is:

$$T_{P1_{RING}} = (P-1) \times t_{SU} + (P-1) \times P \times T_{TR} + (P-1) \times P \times T_{COMP} \quad (4.4)$$

where T_{TR} is the transmission time for a single shooting patch data, T_{COMP} is the time to compare two patch entries of the arrays. Note that, only P comparisons are enough to select the maximum P entries in the two sorted arrays.

On the hypercube topology, the algorithm given above decreases the total complexity by decreasing both the total volume of communication and the comparison computation. The hypercube algorithm requires the following complexity for the communication step:

$$T_{P1_{HYPER}} = d \times t_{SU} + d \times P \times T_{TR} + d \times P \times T_{COMP} \quad (4.5)$$

where $d = logP$. Thus, the hypercube topology performs better than the ring topology. However, computation time required by the shooting patch selection phase is negligible with respect to the other phases discussed in the following sections. It will be seen that the other phases require $O(N)$ time complexity, whereas the shooting patch selection phase requires $O(P^2)$ complexity for the ring and $O(PlogP)$ complexity for the hypercube topology. In general $N \gg P$, hence the performance increase by the hypercube topology over ring topology in this phase does not affect the performance of the overall iteration significantly.

4.2.2 Phase 2: Hemicube Production

In this phase, each processor needs to maintain a hemicube for constructing the form-factor vector corresponding to its local shooting patch. Furthermore, each processor needs to access the whole scene description in order to fill its local hemicube item-buffers corresponding to its local shooting patch. One approach is to replicate the whole patch geometry data in all the processors, hence avoiding interprocessor communication. However, as discussed in the previous chapter, this approach is not suitable for complex scenes with large numbers of patches because of the excessive memory requirement per processor, since $O(N)$ memory is required per processor. Hence, a more valid approach is to evenly decompose whole scene description into P patch data subsets and map each data subset to a distinct processor of the multicomputer, decreasing the memory required per processor to $O(N/P)$.

However, the decomposition of the scene data necessitates global interprocessor communication in this phase since each processor owns only a portion (of size N/P) of the patch database and needs to access the whole database. This requires *circulating* the patch subset of the processors so that each patch data subset visits each of the P processors exactly once.

Note that, only geometry data of the patches are needed for projecting the patches in this phase and communication of the color information is unnecessary. Since the messages can only be sent and received from/into contiguous memory blocks on the iPSC/2, the patch data are divided into geometry and color parts in two arrays as follows:

```
PatchGeometryType = type
  integer           PatchId;
  (x,y,z)           vertex1, vertex2, vertex3;
  RayType           normal;
endtype;

PatchColorType    = type
  (r,g,b)  reflectivity;
  (r,g,b)  delta_radiosity;
  (r,g,b)  radiosity;
  float    area;
endtype;

PatchGeometryArray = array [1..N/P] of PatchGeometryType;
PatchColorArray    = array [1..N/P] of PatchColorType;
```

So, in Phase 2, only the local arrays *PatchGeometryArray* of the processors are circulated. This decreases volume of communication from 23 words to 16 words per triangular patch, obtaining 30% decrease in total volume of communication in Phase 2.

In the following subsections, the algorithms for achieving patch circulation on ring-connected and hypercube-connected multicomputers are presented.

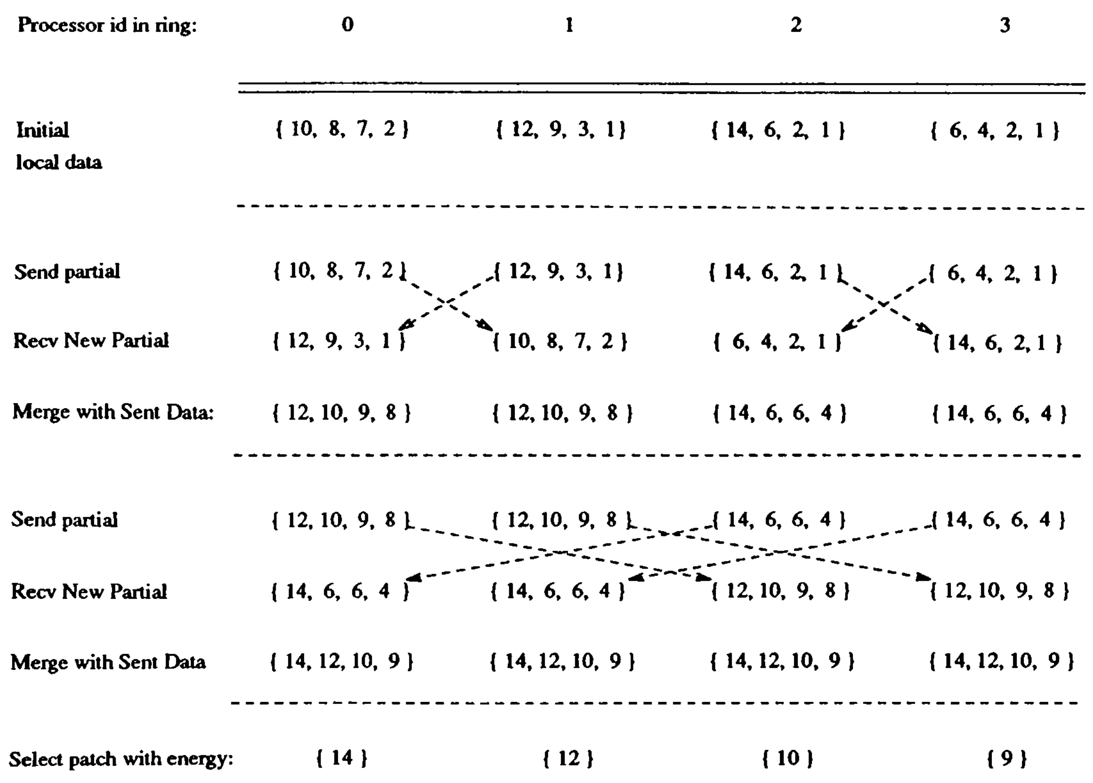
Ring Topology

Patch circulation needed in this phase can be achieved in P concurrent communication steps as follows. In each concurrent step, the current subset of the patch data in the local memory of the processor is projected onto the local hemicube; then this subset is sent to the next processor in the ring, and the new subset is received in a single communication phase. At the end of P concurrent communication steps, each processor completes the projection of all patches onto its local hemicube. Although P-1 communications would be enough for this operation, one more communication is required in order to have the geometry data of local patches in the processors' local memory for maintaining the consistency of geometry and color data for rendering and subsequent iterations.

Figure 4.5 shows the algorithm for the ring topology, and Figure 4.6 illustrates the execution the algorithm on a ring with 4 processors. In this figure, P_i denotes the i^{th} subset of the patch geometry data which corresponds to the local patch data set of processor i. H_i^J denotes that the local hemicube of processor i has been filled by the local patch data of the set of processors J.

Hypercube Topology

For MIMD hypercubes, the patch circulation can be achieved by embedding the ring onto the hypercube topology using Gray Code ordering scheme, as discussed in the previous chapter. Note that, embedding the ring onto the hypercube topology makes use of different channels for interconnecting the processors, not allowing the algorithm to run on SIMD hypercubes. An efficient circulation scheme for SIMD hypercubes can be achieved with the use of exchange sequence X_q, defined as [17]:



Processor id in ring:	0	1	2	3
Initial local data	{ 10, 8, 7, 2 }	{ 12, 9, 3, 1 }	{ 14, 6, 2, 1 }	{ 6, 4, 2, 1 }
Send partial	{ 10, 8, 7, 2 }	{ 12, 9, 3, 1 }	{ 14, 6, 2, 1 }	{ 6, 4, 2, 1 }
Recv New Partial	{ 12, 9, 3, 1 }	{ 10, 8, 7, 2 }	{ 6, 4, 2, 1 }	{ 14, 6, 2, 1 }
Merge with Sent Data:	{ 12, 10, 9, 8 }	{ 12, 10, 9, 8 }	{ 14, 6, 6, 4 }	{ 14, 6, 6, 4 }
Send partial	{ 12, 10, 9, 8 }	{ 12, 10, 9, 8 }	{ 14, 6, 6, 4 }	{ 14, 6, 6, 4 }
Recv New Partial	{ 14, 6, 6, 4 }	{ 14, 6, 6, 4 }	{ 12, 10, 9, 8 }	{ 12, 10, 9, 8 }
Merge with Sent Data	{ 14, 12, 10, 9 }	{ 14, 12, 10, 9 }	{ 14, 12, 10, 9 }	{ 14, 12, 10, 9 }
Select patch with energy:	{ 14 }	{ 12 }	{ 10 }	{ 9 }

Figure 4.4. Example global shooting patch selection on hypercube topology

```
FillHemicube
begin
    nextnode = ( mynode + 1 ) mod P;
    for i=1 to P do
        Project PatchGeometry onto local Hemicube;
        send PatchGeometry to nextnode;
        receive new data into PatchGeometry;
    endfor
end
```

Figure 4.5. Algorithm for Hemicube Production on Ring Topology

$$X_1 = 0 \tag{4.6}$$
$$X_q = X_{q-1}(q-1)X_{q-1}, q > 1 \tag{4.7}$$

For example $X_2 = \{0,1,0\}, X_3 = \{0,1,0,2,0,1,0\}$. This exchange sequence specifies the bidirectional channel to be used for interprocessor communications in each concurrent communication step. In fact, the exchange sequence X_q states that the hypercube of dimension q is divided into two sub-hypercubes (represented by X_{q-1}'s), first the local patch data of the processor is circulated in the subcube in which the processor is located (first X_{q-1} in Equation 4.7), then the patch data are swapped between the corresponding (neighbour) processors in the two subcubes along channel $(q-1)$, and finally the patch data subset is circulated in the other subcube (second X_{q-1} in Equation 4.7).

The SIMD patch circulation scheme for Phase 2 is listed in Figure 4.7. In this algorithm, $f(q,i)$ gives the i^{th} element on the exchange sequence for X_q. For example, $f(2,0) = 0$, $f(2,1) = 1$, $f(3,1) = 1$, $f(3,3) = 2$. The function $f(q,i)$ values are initialized for the values q and i using the recursive equations for definition of X_q in a local array.

Note that (P-1) communication steps are enough for circulating all the patches so that each subset visits all the processors. One more communication along channel $q-1$ is required to have all the patch subsets in their old processors for maintaining patch data consistency.

An example patch circulation on an eight processor hypercube is illustrated in Figure 4.8.

Performance Analysis of Phase 2

Note that the parallel algorithms given for the ring and hypercube topology require P number of concurrent communications and a total of N volume of concurrent communications. Hence, the efficiency of this phase is independent of the interconnection topology of the processors, so the performance of this phase does not degrade with simple topologies. It follows that the parallel complexity of Phase 2 is:

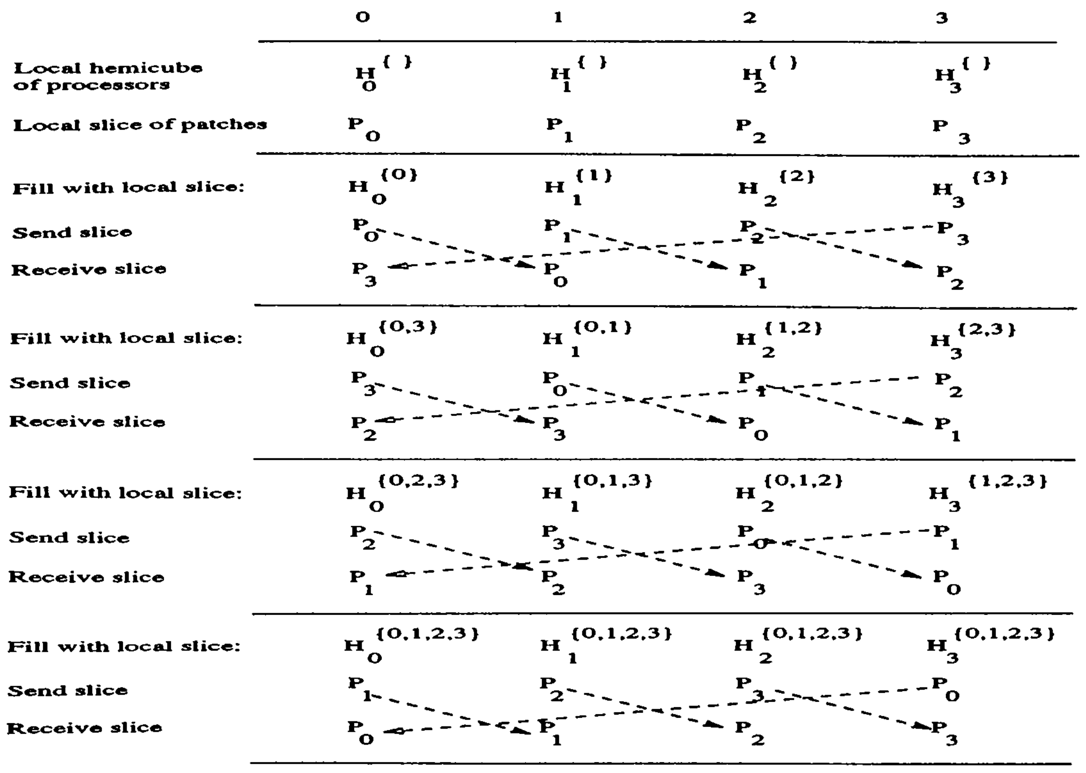


Figure 4.6. Patch Circulation on a Ring with 4 Processors

```
FillHemicube
begin
    for i=1 to P do
        Project PatchGeometry onto local Hemicube;
        dnode = mynode $\oplus$ $2^{f[q,i]}$;
        send PatchGeometry to dnode;
        receive new data into PatchGeometry;
    endfor
end
```

Figure 4.7. Algorithm for Hemicube Production on Hypercube Topology

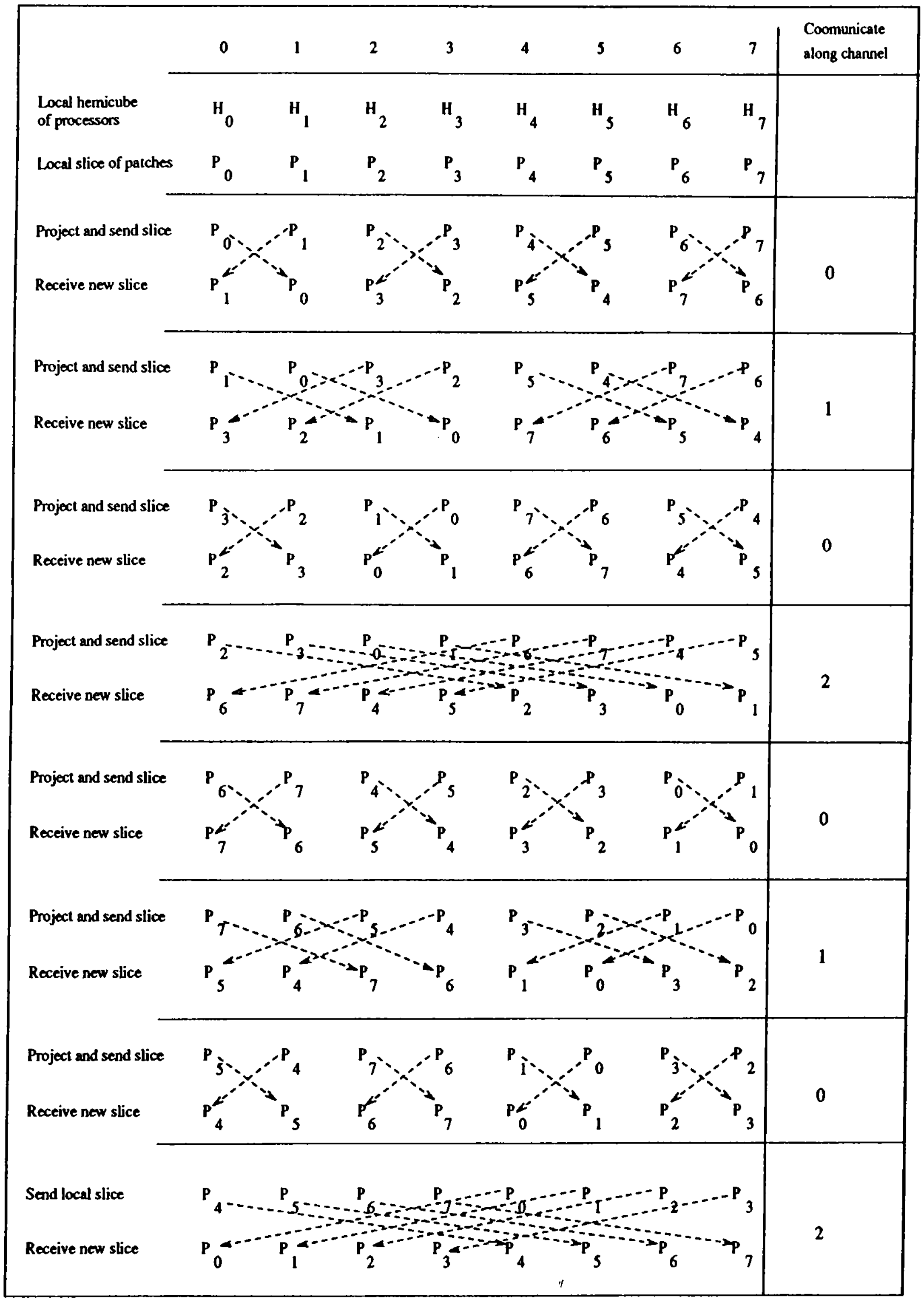

0 1 2 3 4 5 6 7
Coomunicate along channel
Local hemicube of processors
H_0 H_1 H_2 H_3 H_4 H_5 H_6 H_7
Local slice of patches
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
Project and send slice
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
Receive new slice
P_1 P_0 P_3 P_2 P_5 P_4 P_7 P_6
0
Project and send slice
P_1 P_0 P_3 P_2 P_5 P_4 P_7 P_6
Receive new slice
P_3 P_2 P_1 P_0 P_7 P_6 P_5 P_4
1
Project and send slice
P_3 P_2 P_1 P_0 P_7 P_6 P_5 P_4
Receive new slice
P_2 P_3 P_0 P_1 P_6 P_7 P_4 P_5
0
Project and send slice
P_2 P_3 P_0 P_1 P_6 P_7 P_4 P_5
Receive new slice
P_6 P_7 P_4 P_5 P_2 P_3 P_0 P_1
2
Project and send slice
P_6 P_7 P_4 P_5 P_2 P_3 P_0 P_1
Receive new slice
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
0
Project and send slice
P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0
Receive new slice
P_5 P_4 P_7 P_6 P_1 P_0 P_3 P_2
1
Project and send slice
P_5 P_4 P_7 P_6 P_1 P_0 P_3 P_2
Receive new slice
P_4 P_5 P_6 P_7 P_0 P_1 P_2 P_3
0
Send local slice
P_4 P_5 P_6 P_7 P_0 P_1 P_2 P_3
Receive new slice
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
2

Figure 4.8. Patch Circulation on a Three-Dimensional Hypercube

$$\begin{aligned} T_{P2} &= Pt_{SU} + P(N/P)T_{TR} + P(N/P)T_{PRO} \qquad (4.8) \\ &= Pt_{SU} + NT_{TR} + NT_{PRO} \qquad (4.9) \end{aligned}$$

Here, t_{SU} represents the message start-up overhead or the message latency, T_{TR} is the time taken for the transmission of a single patch geometry, T_{PRO} is the average time taken to project and scan-convert one patch onto a hemicube and N is the total number of patches in the scene.

There are two crucial factors that affect the efficiency of the parallelization in this phase: load imbalance and communication overhead. Note that, the parallel complexity given in Eq. 4.9 assumes a perfect load balance among processors. Mapping equal number of patches to each processor achieves balanced communication volume between successive processors in the ring. Furthermore, as will be discussed later, it achieves perfect load balance among processors in the parallel light distribution phase (Phase 4). However, this mapping may not achieve computational balance in the parallel hemicube production phase (Phase 2).

The complexity of the projection of an individual patch onto a hemicube depends on several geometric factors. Recall that, each patch passes through a projection pipeline consisting of visibility test, clipping, perspective projection and scan-conversion. A patch which is not visible by the shooting patch requires much less computation compared to a visible patch since it leaves the projection pipeline in a very early stage. The complexity of the scan-conversion stage for a particular patch depends strongly on the distance and the orientation of that patch with respect to the shooting patch. That is, a patch with larger projection area on a hemicube requires more scan-conversion computation than a patch with a smaller projection area. As is mentioned earlier, each iteration of the proposed algorithm consists of P concurrent steps. At each step, different processors concurrently perform the projection of different sets of patches onto different hemicubes. Hence, the decomposition scheme should be carefully selected in order to maintain the computational load balance in this phase of the algorithm.

Two possible decomposition schemes are *tiled* and *scattered* decompositions. In *tiled* decomposition, the neighbouring patches are stored in the local memory of the same processor. This type of decomposition can be achieved in the following way: assuming that the patches that belong to the same object are supplied consecutively, the first N/P patches are stored in processor 0, the next N/P patches are allocated to processor 1, etc. At the end of the decomposition, each processor stores almost equal number of patches in its local memory. In *scattered* decomposition, the neighbouring patches are stored in different processors, therefore the patches that belong to an object are shared by different processors. Scattered decomposition can be achieved in the following way: again assuming that the neighbouring patches that belong to the same object are supplied consecutively, the incoming patches are allocated to the processors in a round-robin fashion. That is, the first patch is allocated to processor 0, the next to processor 1, etc. When P patches are allocated, the next incoming patch is allocated to processor 0, and this process continues. When the decomposition is completed, (N *mod* P) processors store $\lceil N/P \rceil$ patches, while the remaining processors store $\lfloor N/P \rfloor$ patches in their local memories. Figure 4.9 illustrates the scattered and tiled decomposition of a simple scene consisting of four faces of a room. The numbers shown inside the patches indicate *id*'s of the processors that store them in their local memory.

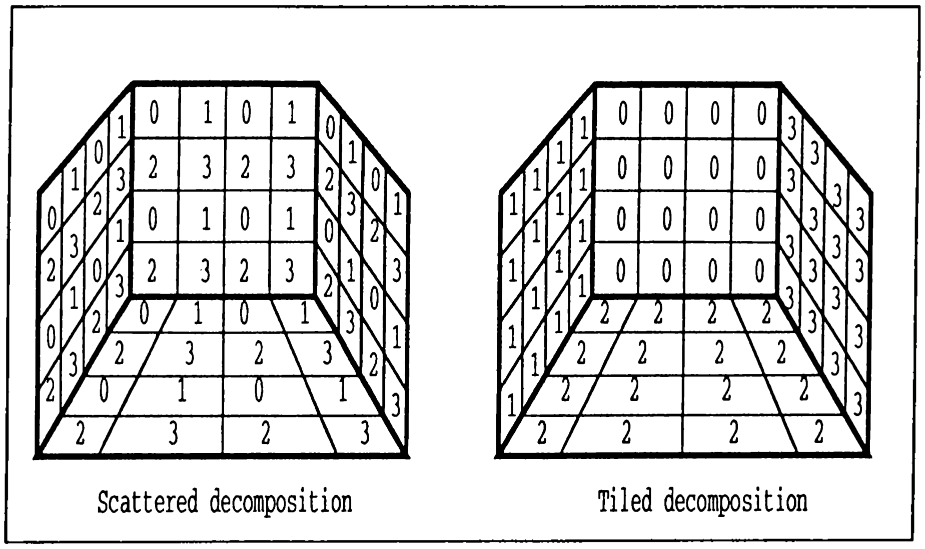


Figure 4.9. Scattered and Tiled Decomposition

Assuming that neighbour patches require almost equal amount of computation for projection on different hemicubes, the scattered decomposition is expected to produce patch partitions requiring almost equal amount of computations in Phase 2. So, it can be expected that the scattered decomposition achieves much better load balance than the tiled decomposition in Phase 2.

Communication overhead in this phase consists of two components: number of communications and volume of communications. Each concurrent communication step adds a fixed message set-up time overhead t_{SU} to the parallel algorithm. In medium grain multicomputers (e.g. Intel's iPSC/2 hypercube) t_{SU} is substantially greater than the transmission time t_{TR} where t_{TR} denotes the time taken for the transmission of a single word. For example, $t_{SU} \approx 550\mu sec$ whereas $t_{TR} \approx 1.44\mu sec$ per word in iPSC/2. Note that, communication of an individual patch geometry involves the transmission of 3 floating point words for the vertices of the triangular patches, 3 words for their normal and one word for the patch id, adding to 52 bytes (i.e. $T_{TR} = 13\ t_{TR}$ in Eq. 4.9). However, as seen in Eq. 4.9, the total number of concurrent communications at each iteration is equal to the number of processors P, whereas the total volume of communication is equal to the number of patches N. Hence, the set-up time overhead can be considered as negligible for complex scenes ($N \gg P$). Then, assuming a perfect load balance, efficiency of Phase 2 can be expressed as:

$$E_{P2} = \frac{1}{P}\frac{PNT_{PRO}}{NT_{PRO} + Pt_{SU} + NT_{TR}} \approx \frac{NT_{PRO}}{NT_{PRO} + NT_{TR}} \quad (4.10)$$

$$= \frac{T_{PRO}}{T_{PRO} + T_{TR}} = \frac{1}{1 + T_{TR}/T_{PRO}} \quad (4.11)$$

since one iteration of the parallel algorithm is computationally equivalent to P iterations of the sequential algorithm. Equation 4.11 means that projection of an individual patch onto a hemicube involves the communication of its geometry data as an overhead. As is seen in Eq. 4.11, the overall efficiency of this phase only depends on the ratio T_{TR}/T_{PRO} for sufficiently large N/P. For example, efficiency is expected to increase with increasing patch areas and increasing hemicube resolution, since the granularity of a projection computation increases with these factors.

Note that, the algorithms for Phase 2 for the ring and hypercube topologies illustrated in Figures 4.5 and 4.7 assume synchronous communication. In this *tightly coupled* communication type, during the circulation, when a processor finishes projection of its current slice it receives at a single iteration, it sends its current patch subset to the receiver processor, and it waits for the next patch subset for projecting onto its local hemicube. If the sender processor has not finished projecting the next subset yet, then this processor becomes idle. This problem can be eliminated by using a *loosely-coupled* communication scheme with asynchronous send and receive operations. In *asynchronous* *send* command, the processor continues to the next instruction, where a hardware router in the processor copies the addressed data to the bidirectional link concurrently. In *asynchronous receive* command, if the message has arrived, the message data is copied from the system buffer to memory address pointed; otherwise the processor continues to the next instruction and the received messages thereafter are received into the system message buffers. Also there is a utility **msgwait** to test if the message has been received by the processor and is ready in the system buffer.

The new approach is based on the observation that, when the processors finish projecting the first half of their current subset of patches during each step, that half will not be used by the processors. Hence, the processors issue a synchronous send operation to their receiver processors, and issue an asynchronous receive operation from their sender processors to their already projected half. Thus, when a processor finishes projecting the whole subset of the patch data onto its local hemicube, it has received the first half of the next subset to its local memory, so it can continue projecting the received half while issuing an asynchronous receive command for the second half. In this new approach, in order for a processor to become idle, the projection of half of a subset of the patch data onto a local processor's hemicube should take more time than the projection of two halves of another patch subset onto another local hemicube, which is not usual.

4.2.3 Phase 3: Form-Factor Vector Computation

In this phase, each processor can concurrently compute the form-factor vector corresponding to its shooting patch using its local hemicube item-buffers constructed in the previous phase. This phase requires no interprocessor communication. Local form-factor vector computations involved in this phase require scanning all hemicube item-buffer entries. Hence, perfect load balance is easily achieved since each processor maintains a hemicube of equal resolution.

4.2.4 Phase 4: Contribution Computation

At the end of Phase 3, each processor holds a form-factor vector corresponding to its shooting patch. In this phase, each processor should compute the light contributions from all P shooting patches to its local patches. Hence, each processor needs all form-factor vectors. Thus, this phase necessitates global interprocessor communication since each processor owns only a single form-factor vector.

We introduce a vector notation for the sake of clarity of the presentation of the algorithms discussed in this section. Let $\mathbf{X_k}$ denote the k^{th} slice of a global vector $\mathbf{X}$ assigned to processor k. For example, each processor k can be considered as storing the k^{th} slice of the global array of records representing the whole patch geometry. In this notation, each processor k is responsible for computing the k^{th} slice $\Delta\mathbf{R_k}$ of the global contribution vector $\Delta\mathbf{R}$ for updating the k^{th} slices $\mathbf{B_k}$ and $\Delta\mathbf{B_k}$ of the global radiosity and delta radiosity vectors $\mathbf{B}$ and $\Delta\mathbf{B}$, respectively. The notation used to label the P distinct form-factor vectors maintained by P processors is slightly different. In this case, $\mathbf{F}^{\ell}$ denotes the form-factor vector computed by processor ℓ and $\mathbf{F}^{\ell}_{\mathbf{k}}$ denotes the k^{th} slice of the local form-factor of processor ℓ. As is seen in Eq. 2.17, red, green and blue reflectivity values $r_i(r,g,b)$ and the patch area A_i of each patch i are needed as three ratios $r_i(r,g,b)/A_i$ in the computation of radiosity and delta radiosity contribution computation. Hence, each processor computes three constants $r_i(r,g,b)/A_i$ for each local patch i during the preprocessing. In vector notation, each processor k can be considered as holding the k^{th} slice $\mathbf{r_k}(r,g,b)$ of the global vector $\mathbf{r}(r,g,b)$.

Hence, in vector notation, each processor k, for $k = 0,1,...,P-1$, is responsible for computing

$$\begin{aligned}
\Delta\mathbf{R_k}(r,g,b) &= \sum_{\ell=0}^{P-1}(\Delta B^{\ell}_{s}(r,g,b)A^{\ell}_{s})\mathbf{r_k}(r,g,b) \times \mathbf{F}^{\ell}_{\mathbf{k}} && (4.12)\\
\mathbf{B_k}(r,g,b) &= \mathbf{B_k}(r,g,b) + \Delta\mathbf{R_k}(r,g,b) && (4.13)\\
\Delta\mathbf{B_k}(r,g,b) &= \Delta\mathbf{B_k}(r,g,b) + \Delta\mathbf{R_k}(r,g,b) && (4.14)
\end{aligned}$$

where $\Delta B^{\ell}_{s}(r,g,b)$ and A^{ℓ}_{s} denote the delta radiosity values for three color-bands and the area of the shooting patch of processor ℓ. In Eq. 4.12, "$\times$" denotes the element-by-element multiplication of two column vectors. As is seen in Eq. 4.13 and 4.14, radiosity and delta radiosity computations are local vector additions which do not require any interprocessor communication. It is the contribution computation phase (Eq. 4.12), which requires global interaction. Equation 4.12 can be rewritten by factoring out the $\mathbf{r_k}$ vector as

$$\Delta \mathbf{R}_{\mathbf{k}}(r,g,b) = \mathbf{r}_{\mathbf{k}}(r,g,b) \times \mathbf{U}_{\mathbf{k}}(r,g,b) \tag{4.15}$$

where,

$$\mathbf{U}_{\mathbf{k}}(r,g,b) = \sum_{\ell=0}^{P-1} \mathbf{U}_{\mathbf{k}}^{\ell}(r,g,b) \tag{4.16}$$

$$\mathbf{U}_{\mathbf{k}}^{\ell}(r,g,b) = \Delta B_s^{\ell}(r,g,b) A_s^{\ell} \mathbf{F}_{\mathbf{k}}^{\ell} \tag{4.17}$$

Note that, the notation used to label the **U** vectors is similar to that of the **F** vectors since the P **U** vectors, of sizes N/P, are concurrently computed by P processors. That is, $\mathbf{U}_{\mathbf{k}}^{\ell}(r,g,b)$ represents the contribution vector of the shooting patch of processor ℓ to the local patches of processor k omitting the multiplications with the $r_i(r,g,b)/A_i$ coefficients. Hence, $\mathbf{U}_{\mathbf{k}}(r,g,b)$ represents the total contribution vector of all P shooting patches to the local patches of processor k.

In the following paragraphs, the ring and hypercube algorithms for performing this phase are presented.

Ring Topology

The first approach discussed in this work is very similar to the implementation proposed by Chalmers *et al.* [12]. In their implementation, each processor ℓ ***broadcasts*** a packet consisting of the delta radiosities, area and the form-factor vector of its shooting patch. Each processor k, upon receiving a packet $\{\ \Delta B_s^{\ell}, A_s^{\ell}, \mathbf{F}^{\ell}\ \}$, computes a local contribution vector $\mathbf{U}_{\mathbf{k}}^{\ell}(\mathbf{r},\mathbf{g},\mathbf{b})$ by performing a local scalar vector product for each color (Eq. 4.17) and ***accumulates*** this vector to its local $\mathbf{U}_{\mathbf{k}}(r,g,b)$ vector by performing a local vector addition operation (Eq. 4.16). However, multiple broadcast operations are expensive and may cause excessive congestion in ring interconnection topologies. In this work, indicated packets are circulated in a synchronous manner, similar to the patch circulation discussed for Phase 2. Form-factor vector circulation consists of *P-1* concurrent communication steps. In each step, each processor sends its current packet to the next processor in the ring, and receives a new packet from the previous processor in the ring. Between each successive communication steps, each processor concurrently performs the contribution vector accumulation computations (Eqs. 4.17 and 4.16) corresponding to its current packet. At the end of P-1 concurrent communication steps, each processor k accumulates its total contribution vector $\mathbf{U_k}(r,g,b)$. Then, each processor k can concurrently compute its local $\mathbf{\Delta R_k}(r,g,b)$ vector by performing a local element-by-element vector multiplication for each color (Eq. 4.15). Figure 4.10 illustrates the pseudocode for this approach.

```
/* $\Delta B_s(r,g,b)$ : delta radiosity of local shooting patch;
   $A_s$ : area of local shooting patch;
   F : local form-factor vector (of size $N$);
   U, ΔR, B, ΔB are local vectors (of size $N/P$);*/

   $nextnode$ = (mynode + 1) mod P;
   $prevnode$ = (mynode - 1) mod P;
   $k$ = mynode;
   $\mathbf{U}(r,g,b)$ = $\Delta B_s(r,g,b,)A_s\mathbf{F_k}$

   for i=1 to P-1 do
       send ($\Delta B_s(r,g,b)$,$A_s$, F) to processor $nextnode$;
       receive into ($\Delta B_s(r,g,b)$,$A_s$, F) from processor $prevnode$;
       $\mathbf{U}(r,g,b)$ = $\mathbf{U}(r,g,b)$ + $\Delta B_s(r,g,b)A_s\mathbf{F_k}$;
   endfor
   $\mathbf{\Delta R}(r,g,b)$ = $\mathbf{r}(r,g,b)$ * $\mathbf{U}(r,g,b)$;
   $\mathbf{B}(r,g,b)$ = $\mathbf{B}(r,g,b)$ + $\mathbf{\Delta R}(r,g,b)$;
   $\mathbf{\Delta B}(r,g,b)$ = $\mathbf{\Delta B}(r,g,b)$ + $\mathbf{\Delta R}(r,g,b)$;
end
```

Figure 4.10. The Form-Factor Vector Circulation Scheme for the Ring Topology

It is obvious that perfect load balance in this phase can easily be achieved by mapping equal number of patches to each processor. Hence, the parallel complexity of Phase 4 using the described *form-factor vector circulation* scheme, is:

$$T_{P4} = (P-1)t_{SU} + (P-1)Nt_{tr} + P(N/P)T_{CONTR} + (N/P)T_{UPD} \quad (4.18)$$

$$= (P-1)t_{SU} + (P-1)Nt_{tr} + NT_{CONTR} + (N/P)T_{UPD} \quad (4.19)$$

Here, t_{tr} is the time taken to transmit a single floating point word, T_{CONTR}

is the time taken to compute and accumulate a single contribution value, and T_{UPD} is the time taken to update a single radiosity and delta radiosity value using the corresponding entry of a local $\mathbf{U_k}$ vector.

Note that, in this scheme, processors accumulate the contributions for their local patches during the circulation of form-factor vectors. Hence, as is also seen in Eq. 4.19, this scheme necessitates high volume of communication ($(P-1)N$ words) since whole form-factor vectors of sizes N are concurrently communicated at each communication step. However, as is also seen in Eq. 4.16, each processor k needs only the k^{th} slices (of sizes N/P) of the form-factor vectors it receives during the circulation. That is, form-factor circulation scheme involves the circulation of redundant information. In this work, we propose an efficient scheme which avoids this redundancy in the interprocessor communication. In the proposed scheme, partial contribution computation results ($\mathbf{U^\ell_k}(r,g,b)$ vectors of sizes N/P) are circulated instead of the form-factor vectors (of sizes N). Hence, each processor effectively accumulates the contributions of its local shooting patch to all other processors' local patches during the circulation of the partial contribution computation results.

In a straightforward implementation of the proposed new scheme, each processor k first constructs the contribution vector ($\mathbf{U^k_k}(r,g,b)$) of its shooting patch to its local patches, and initiates the circulation of contribution vectors. After the i^{th} concurrent communication step in the circulation, processor k constructs the $\mathbf{U^k_{(k-i)modP}}(r,g,b)$ vector using the slice $\mathbf{F^k_{(k-i)modP}}$ of its local form-factor vector $\mathbf{F^k}$, and accumulates this vector to the current partial contribution vector. At the end of $P-1$ concurrent communication steps, each processor k holds the final contribution vector $\mathbf{U_{(k+1)modP}}(r,g,b)$ of the next processor in the ring. Hence, one more communication is needed in order to return the final contribution vectors to their "home" processors. However, this communication step can avoided by each processor k constructing $\mathbf{U^k_{(k-1)modP}}(r,g,b)$ vector in the initialization phase and accumulating the $\mathbf{U^k_{(k-i-1)modP}}(r,g,b)$ vector at step i. Figure 4.12 illustrates the operation of the proposed scheme on a ring with 4 processors.

The proposed circulation scheme also preserves the perfect load balance, if exactly equal number of patches are mapped to each processor. Hence, the proposed circulation scheme reduces the overall parallel complexity of Phase 4 to

```
/* $\Delta B_s(r,g,b)$ :  delta radiosity of local shooting patch;
   $A_s$ :  area of local shooting patch;
   $\mathbf{F}$ :  local form-factor vector (of size $N$);
   U, ΔR, B, ΔB are local vectors (of size $N/P$); */
   nextnode = (mynode + 1) mod P;
   prevnode = (mynode - 1) mod P;
   k = mynode;        $\mathrm{U}(r,g,b) = \Delta B_s(r,g,b) A_s \mathbf{F}_{\mathrm{prevnode}}$;

   for i=1 to P-1 do
       send $\mathrm{U}(r,g,b)$ to processor nextnode;
       receive into $\mathrm{U}(r,g,b)$ from processor prevnode;
       $\mathrm{U}(r,g,b) = \mathrm{U}(r,g,b) + \Delta B_s(r,g,b) A_s\ \mathbf{F}_{(k-i-1) \mathrm{mod} P}$;
   endfor
   $\Delta\mathrm{R}(r,g,b) = \mathrm{r}(r,g,b) * \mathrm{U}(r,g,b)$;
   $\mathrm{B}(r,g,b) = \mathrm{B}(r,g,b) + \Delta\mathrm{R}(r,g,b)$;
   $\Delta\mathrm{B}(r,g,b) = \Delta\mathrm{B}(r,g,b) + \Delta\mathrm{R}(r,g,b)$;
```

Figure 4.11. The Contribution Vector Circulation Scheme for the Ring Topology

$$\begin{aligned} T_{P4} &= (P-1)t_{SU} + 3(P-1)(N/P)t_{tr} + P(N/P)T_{CONTR} + (N/P)T_{UPD} && (4.20) \\ &= (P-1)t_{SU} + 3\frac{P-1}{P}Nt_{tr} + NT_{CONTR} + \frac{N}{P}T_{UPD} && (4.21) \end{aligned}$$

Note that, the constant 3 appears as a coefficient in "t_{tr}" term since each entry of an individual $\mathbf{U}_k^1$ vector consists of 3 contribution values for 3 color-bands. Hence, the proposed circulation scheme reduces the total concurrent communication volume in Phase 4 by an asymptotical factor of $P/3$ for $P > 3$.

Hypercube Topology

The algorithm proposed for the SIMD hypercubes is similar to that for the ring topology. This phase again requires the exchange sequence introduced for Phase 2 in order to circulate the partial contribution computation results..

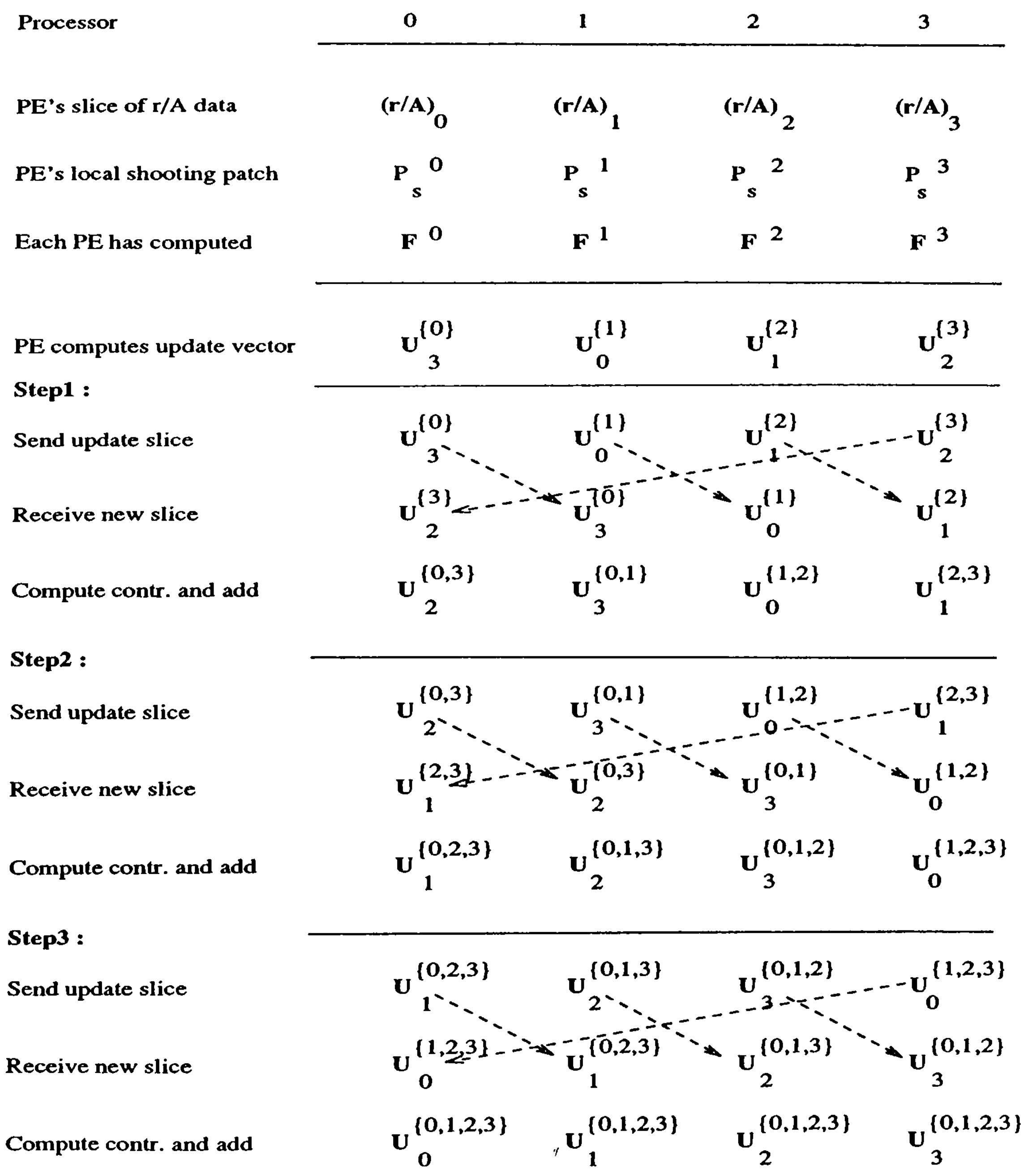
The algorithm for the hypercube topology is presented in Figure 4.13 and an example execution of Phase 2 on hypercube topology is illustrated in Figure 4.14.

The proposed hypercube algorithm also preserves the perfect load balance, if exactly equal number of patches are mapped to each processor. Hence, the parallel complexity of the proposed circulation scheme for Phase 4 on hypercube topology is equal to that of the ring topology given in Eq. 4.21.

4.3 Experimental Results

The proposed schemes are implemented on an Intel's iPSC/2 hypercube multicomputer. The performance of the system is tested on a ring with 1, 2, 4, 6, 8, 10, 12, 14 and 16 processors. The hypercube algorithm has the same time complexity and task allocation scheme, therefore these performance results are valid for the hypercube topology.

The form factors are computed using hemicubes of constant resolution $50 \times 100 \times 100$. The proposed parallel algorithms are experimented for five different scenes with 856, 1412, 3424, 5648 and 8352 patches.



Processor	0	1	2	3
PE's slice of r/A data	$(r/A)_0$	$(r/A)_1$	$(r/A)_2$	$(r/A)_3$
PE's local shooting patch	P_s^0	P_s^1	P_s^2	P_s^3
Each PE has computed	F^0	F^1	F^2	F^3
PE computes update vector	$U_3^{\{0\}}$	$U_0^{\{1\}}$	$U_1^{\{2\}}$	$U_2^{\{3\}}$
Step1 :				
Send update slice	$U_3^{\{0\}}$	$U_0^{\{1\}}$	$U_1^{\{2\}}$	$U_2^{\{3\}}$
Receive new slice	$U_2^{\{3\}}$	$U_3^{\{0\}}$	$U_0^{\{1\}}$	$U_1^{\{2\}}$
Compute contr. and add	$U_2^{\{0,3\}}$	$U_3^{\{0,1\}}$	$U_0^{\{1,2\}}$	$U_1^{\{2,3\}}$
Step2 :				
Send update slice	$U_2^{\{0,3\}}$	$U_3^{\{0,1\}}$	$U_0^{\{1,2\}}$	$U_1^{\{2,3\}}$
Receive new slice	$U_1^{\{2,3\}}$	$U_2^{\{0,3\}}$	$U_3^{\{0,1\}}$	$U_0^{\{1,2\}}$
Compute contr. and add	$U_1^{\{0,2,3\}}$	$U_2^{\{0,1,3\}}$	$U_3^{\{0,1,2\}}$	$U_0^{\{1,2,3\}}$
Step3 :				
Send update slice	$U_1^{\{0,2,3\}}$	$U_2^{\{0,1,3\}}$	$U_3^{\{0,1,2\}}$	$U_0^{\{1,2,3\}}$
Receive new slice	$U_0^{\{1,2,3\}}$	$U_1^{\{0,2,3\}}$	$U_2^{\{0,1,3\}}$	$U_3^{\{0,1,2\}}$
Compute contr. and add	$U_0^{\{0,1,2,3\}}$	$U_1^{\{0,1,2,3\}}$	$U_2^{\{0,1,2,3\}}$	$U_3^{\{0,1,2,3\}}$

Figure 4.12. Contribution computation on a Ring with 4 Processors

```
/* $\Delta B_s(r,g,b)$ :  delta radiosity of local shooting patch;
   $A_s$ :  area of local shooting patch;
   $\mathbf{F}$ :  local form-factor vector (of size $N$);
   ix :  Holds the current processor slice computed at each iteration
   U, $\Delta$R, B, $\Delta$B are local vectors (of size $N/P$); */
   ix = mynode $\oplus$ $2^{d-1}$;
   U$(r,g,b)$ = $\Delta B_s(r,g,b) A_s \mathbf{F}_{ix}$;

   for i=0 to P-1 do
       dnode = mynode $\oplus$ $2^{f[q,i]}$;
       send U$(r,g,b)$ to processor dnode;
       receive into U$(r,g,b)$ from processor dnode;
       ix = ix $\oplus$ $2^{f[q,i]}$;
       U$(r,g,b)$ = U$(r,g,b)$ + $\Delta B_s(r,g,b) A_s$ $\mathbf{F}_{ix}$;
   endfor
   $\Delta$R$(r,g,b)$ = r$(r,g,b)$ * U$(r,g,b)$;
   B$(r,g,b)$ = B$(r,g,b)$ + $\Delta$R$(r,g,b)$;
   $\Delta$B$(r,g,b)$ = $\Delta$B$(r,g,b)$ + $\Delta$R$(r,g,b)$;
```

Figure 4.13. The Contribution Vector Circulation Scheme for the Hypercube Topology

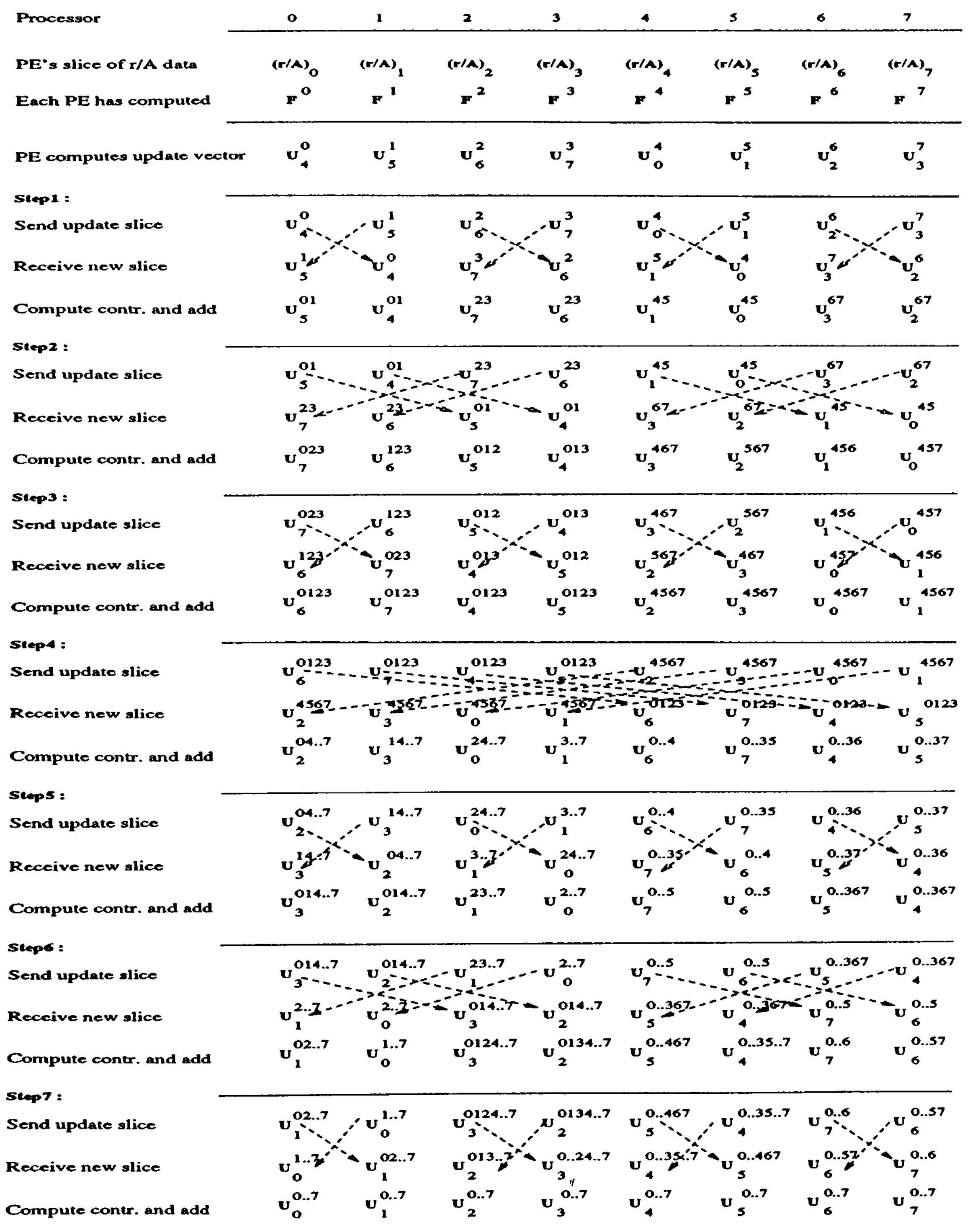

Processor 0 1 2 3 4 5 6 7
PE's slice of r/A data $(r/A)_0$ $(r/A)_1$ $(r/A)_2$ $(r/A)_3$ $(r/A)_4$ $(r/A)_5$ $(r/A)_6$ $(r/A)_7$
Each PE has computed F^0 F^1 F^2 F^3 F^4 F^5 F^6 F^7
PE computes update vector U^0_4 U^1_5 U^2_6 U^3_7 U^4_0 U^5_1 U^6_2 U^7_3
Step1 :
Send update slice U^0_4 U^1_5 U^2_6 U^3_7 U^4_0 U^5_1 U^6_2 U^7_3
Receive new slice U^1_5 U^0_4 U^3_7 U^2_6 U^5_1 U^4_0 U^7_3 U^6_2
Compute contr. and add U^{01}_5 U^{01}_4 U^{23}_7 U^{23}_6 U^{45}_1 U^{45}_0 U^{67}_3 U^{67}_2
Step2 :
Send update slice U^{01}_5 U^{01}_4 U^{23}_7 U^{23}_6 U^{45}_1 U^{45}_0 U^{67}_3 U^{67}_2
Receive new slice U^{23}_7 U^{23}_6 U^{01}_5 U^{01}_4 U^{67}_3 U^{67}_2 U^{45}_1 U^{45}_0
Compute contr. and add U^{023}_7 U^{123}_6 U^{012}_5 U^{013}_4 U^{467}_3 U^{567}_2 U^{456}_1 U^{457}_0
Step3 :
Send update slice U^{023}_7 U^{123}_6 U^{012}_5 U^{013}_4 U^{467}_3 U^{567}_2 U^{456}_1 U^{457}_0
Receive new slice U^{123}_6 U^{023}_7 U^{013}_4 U^{012}_5 U^{567}_2 U^{467}_3 U^{457}_0 U^{456}_1
Compute contr. and add U^{0123}_6 U^{0123}_7 U^{0123}_4 U^{0123}_5 U^{4567}_2 U^{4567}_3 U^{4567}_0 U^{4567}_1
Step4 :
Send update slice U^{0123}_6 U^{0123}_7 U^{0123}_4 U^{0123}_5 U^{4567}_2 U^{4567}_3 U^{4567}_0 U^{4567}_1
Receive new slice U^{4567}_2 U^{4567}_3 U^{4567}_0 U^{4567}_1 U^{0123}_6 U^{0123}_7 U^{0123}_4 U^{0123}_5
Compute contr. and add $U^{04..7}_2$ $U^{14..7}_3$ $U^{24..7}_0$ $U^{3..7}_1$ $U^{0..4}_6$ $U^{0..35}_7$ $U^{0..36}_4$ $U^{0..37}_5$
Step5 :
Send update slice $U^{04..7}_2$ $U^{14..7}_3$ $U^{24..7}_0$ $U^{3..7}_1$ $U^{0..4}_6$ $U^{0..35}_7$ $U^{0..36}_4$ $U^{0..37}_5$
Receive new slice $U^{14..7}_3$ $U^{04..7}_2$ $U^{3..7}_1$ $U^{24..7}_0$ $U^{0..35}_7$ $U^{0..4}_6$ $U^{0..37}_5$ $U^{0..36}_4$
Compute contr. and add $U^{014..7}_3$ $U^{014..7}_2$ $U^{23..7}_1$ $U^{2..7}_0$ $U^{0..5}_7$ $U^{0..5}_6$ $U^{0..367}_5$ $U^{0..367}_4$
Step6 :
Send update slice $U^{014..7}_3$ $U^{014..7}_2$ $U^{23..7}_1$ $U^{2..7}_0$ $U^{0..5}_7$ $U^{0..5}_6$ $U^{0..367}_5$ $U^{0..367}_4$
Receive new slice $U^{2..7}_1$ $U^{2..7}_0$ $U^{014..7}_3$ $U^{014..7}_2$ $U^{0..367}_5$ $U^{0..367}_4$ $U^{0..5}_7$ $U^{0..5}_6$
Compute contr. and add $U^{02..7}_1$ $U^{1..7}_0$ $U^{0124..7}_3$ $U^{0134..7}_2$ $U^{0..467}_5$ $U^{0..35..7}_4$ $U^{0..6}_7$ $U^{0..57}_6$
Step7 :
Send update slice $U^{02..7}_1$ $U^{1..7}_0$ $U^{0124..7}_3$ $U^{0134..7}_2$ $U^{0..467}_5$ $U^{0..35..7}_4$ $U^{0..6}_7$ $U^{0..57}_6$
Receive new slice $U^{1..7}_0$ $U^{02..7}_1$ $U^{013..7}_2$ $U^{0..24..7}_3$ $U^{0..35..7}_4$ $U^{0..467}_5$ $U^{0..57}_6$ $U^{0..6}_7$
Compute contr. and add $U^{0..7}_0$ $U^{0..7}_1$ $U^{0..7}_2$ $U^{0..7}_3$ $U^{0..7}_4$ $U^{0..7}_5$ $U^{0..7}_6$ $U^{0..7}_7$

Figure 4.14. Contribution computation on a Hypercube with 8 Processors

Table 4.1. Effect of local and global shooting patch selection (in Phase 1) on convergence

N	P	Total Number of shooting patch selections for convergence			
		Local	Global	Percent Decrease	Sequential
856	2	214	210	1.87	225
	4	348	228	24.48	
	6	258	222	13.95	
	8	392	232	40.82	
	10	360	240	22.22	
	12	420	240	42.86	
	14	504	266	47.22	
	16	464	288	39.93	
1412	2	386	378	2.07	377
	4	416	388	6.63	
	6	432	384	11.11	
	8	584	400	31.51	
	10	460	410	10.87	
	12	480	420	12.50	
	14	644	434	32.61	
	16	560	448	22.50	
3424	2	324	312	3.70	323
	4	356	316	11.24	
	6	360	342	5.00	
	8	376	320	14.89	
	10	360	320	11.11	
	12	432	312	27.78	
	14	392	350	10.71	
	16	464	336	27.59	
5648	2	270	242	10.29	248
	4	288	240	16.67	
	6	252	252	0.00	
	8	328	248	24.39	
	10	290	250	13.79	
	12	300	252	16.00	
	14	294	280	4.77	
	16	432	272	37.04	

Table 4.1 illustrates the effect of the local and global shooting patch selection (in Phase 1) on the convergence of the parallel algorithm. As is seen in Table 4.1, global selection substantially increases the convergence rate of the parallel algorithm, even though the scattered decomposition scheme is utilized. The fifth column of Table 4.1 shows the percent decrease in the total number of shooting patch selections when the global selection scheme is used instead of the local selection scheme in Phase 1. As is seen in this column, the advantage of the global selection over the local selection generally increases substantially with the increasing number of processors for each scene geometry experimented in this table. Hence, for large number of processors ($P \geq 8$), global shooting patch selection is recommended in the first phase of the parallel algorithm.

Table 4.2 shows the effect of the decomposition scheme on the performance of the hemicube production phase (Phase 2) of the parallel algorithm. Parallel timings (T_{PAR}) in Table 4.2 denote the parallel hemicube production time per shooting patch. These timings are computed as the execution time of P concurrent hemicube productions divided by P since P hemicubes are concurrently produced for P shooting patches in a single iteration of Phase 2. Sequential timings (T_{SEQ}) in Table 4.2 denote the sequential execution time of a single hemicube production. Efficiency values in Table 4.2 are computed using $Eff = T_{SEQ}/(PT_{PAR})$. An efficiency value denotes quality of a decomposition scheme on load balance. As is seen in Table 4.2, scattered decomposition always achieves better load balance than the tiled decomposition. Note that, as the number of processors increases, load balance quality of the scattered decomposition increases in comparison with that of the tiled decomposition. Furthermore, the performance of the loosely-coupled approach for scattered decomposition is almost always better than the tightly-coupled approach for communication because of the improved load balance. As is also seen in Table 4.2, efficiency of both decomposition schemes decrease with increasing P for a fixed N. This decrease in efficiency is due to the increase in the number of synchronization steps since each synchronization step contributes to the overall load imbalance.

Table 4.3 illustrates the execution times of the distributed contribution vector computation during a single iteration of the parallel algorithm. The last column of Table 4.3 illustrates the percent decrease in the execution times obtained by using contribution vector circulation instead of form-factor vector circulation. Note that, the advantage of the contribution vector circulation over the form-factor circulation increases with increasing P as is expected.

Table 4.2. Effect of the decomposition scheme on the performance of the hemicube production phase (Phase 2) of the parallel algorithm

N	Sequential time(secs)	P	Tiled Decomposition		Scattered Decomposition			
			Tightly-coupled		Tightly-coupled		Loosely-coupled	
			parallel time	parallel efficiency	parallel time	parallel efficiency	parallel time	parallel efficiency
856	5.020	2	3.102	0.809	2.618	0.958	2.625	0.956
		4	1.804	0.696	1.374	0.913	1.361	0.922
		6	1.309	0.639	0.931	0.898	0.921	0.908
		8	0.994	0.631	0.722	0.868	0.698	0.899
		10	0.898	0.559	0.568	0.884	0.560	0.894
		12	0.752	0.556	0.482	0.867	0.471	0.888
		14	0.691	0.519	0.426	0.842	0.409	0.877
		16	0.618	0.508	0.380	0.826	0.358	0.876
1412	6.460	2	3.811	0.848	3.429	0.941	3.425	0.943
		4	2.350	0.687	1.812	0.891	1.805	0.895
		6	1.824	0.590	1.228	0.876	1.219	0.883
		8	1.405	0.575	0.936	0.863	0.919	0.879
		10	1.219	0.530	0.749	0.861	0.743	0.870
		12	1.017	0.529	0.629	0.856	0.621	0.867
		14	0.894	0.516	0.549	0.840	0.536	0.861
		16	0.836	0.483	0.486	0.831	0.470	0.859
3424	13.842	2	8.414	0.822	7.314	0.946	7.308	0.947
		4	4.826	0.717	3.834	0.902	3.828	0.904
		6	3.440	0.671	2.537	0.909	2.534	0.910
		8	2.630	0.658	1.949	0.888	1.946	0.889
		10	2.238	0.619	1.570	0.881	1.567	0.883
		12	1.911	0.604	1.316	0.876	1.313	0.879
		14	1.713	0.577	1.127	0.877	1.122	0.881
		16	1.473	0.587	0.990	0.873	0.987	0.878
5648	20.116	2	11.103	0.905	10.627	0.947	10.543	0.954
		4	6.498	0.774	5.438	0.924	5.432	0.926
		6	4.852	0.690	3.673	0.912	3.667	0.914
		8	3.639	0.690	2.762	0.910	2.758	0.912
		10	3.148	0.639	2.217	0.907	2.213	0.909
		12	2.731	0.614	1.857	0.902	1.852	0.905
		14	2.349	0.612	1.595	0.901	1.591	0.903
		16	2.212	0.568	1.397	0.899	1.391	0.904
8352	28.507	2	16.988	0.839	15.065	0.946	15.033	0.948
		4	9.649	0.739	7.730	0.921	7.694	0.926
		6	6.782	0.701	5.377	0.884	5.375	0.884
		8	5.372	0.663	4.011	0.888	4.008	0.889
		10	4.445	0.641	3.131	0.911	3.117	0.915
		12	3.864	0.615	2.633	0.902	2.621	0.907
		14	3.382	0.602	2.309	0.882	2.296	0.887
		16	2.992	0.595	2.052	0.868	2.037	0.875

Table 4.3. Effect of the circulation scheme on the performance of the light contribution computation phase (Phase 4) of the parallel algorithm

N	P	Contribution Computation Time (secs)		
		form factor vector circulation	Contribution vector circulation	percent decrease
856	2	0.0502	0.0482	3.98
	4	0.0604	0.0576	4.64
	6	0.0660	0.0630	4.55
	8	0.0762	0.0664	12.63
	10	0.0842	0.0702	16.67
	12	0.0912	0.0732	19.74
	14	0.0994	0.0756	23.94
	16	0.1072	0.0784	26.87
1412	2	0.0945	0.0911	3.60
	4	0.1012	0.0962	4.94
	6	0.1114	0.1014	8.98
	8	0.1232	0.1064	13.64
	10	0.1345	0.1102	18.07
	12	0.1452	0.1140	21.49
	14	0.1568	0.1162	25.89
	16	0.1680	0.1184	29.52
3424	2	0.1953	0.1879	3.80
	4	0.2305	0.2202	4.46
	6	0.2628	0.2382	9.36
	8	0.2872	0.2456	14.48
	10	0.3110	0.2520	18.97
	12	0.3348	0.2556	23.66
	14	0.3598	0.2604	27.63
	16	0.3840	0.2640	31.25
5648	2	0.3261	0.3183	2.39
	4	0.3920	0.3680	6.12
	6	0.4374	0.3948	9.74
	8	0.4760	0.4072	14.45
	10	0.5153	0.4150	19.46
	12	0.5556	0.4212	24.19
	14	0.5937	0.4270	28.08
	16	0.6385	0.4303	32.61
8352	2	0.4652	0.4596	1.20
	4	0.5405	0.5081	6.00
	6	0.5946	0.5424	8.78
	8	0.6552	0.5608	14.41
	10	0.7087	0.5703	19.53
	12	0.7645	0.5772	24.50
	14	0.8195	0.5838	28.76
	16	0.8832	0.5888	33.33

Figure 4.15 illustrates the overall efficiency curves of the patch circulation approach. In Fig 4.15, the efficiency curves are constructed using

$$Efficiency = \frac{1}{P}\frac{T_{SEQ}}{T_{PAR}}$$

Here, T_{SEQ} and T_{PAR} denote the execution time taken for the sequential algorithm and the parallel algorithm on P processors, respectively, to converge to the same tolerance value. Note that, global shooting patch selection, scattered decomposition and contribution vector circulation schemes are used in Phases 1, 2 and 4, respectively, in order to obtain utmost parallel performance. As is seen in Fig. 4.15, efficiency decreases with increasing P for a fixed N. There are two main reasons for this decrease in the efficiency. The first one is the slight increase in the load imbalance of the parallel hemicube production phase with increasing P. The second, and the more crucial reason is the modification introduced to the original sequential algorithm for the sake of parallelization. As is discussed in Section 4.2, this modification increases the total number of shooting patch selections required for convergence in comparison with the sequential algorithm. Figure 4.16, which illustrates the normalized efficiency values per single shooting patch computation is presented in order to confirm the latter reason. Figure 4.16 eliminates the effect of the increase in the number of shooting patch selections. Greater efficiency values in Fig. 4.16 than those in Fig. 4.15 reveal the performance degradation in the proposed parallel algorithm due to the increase in the number of shooting patch selections. As is seen in Fig 4.16, the efficiency of the parallel algorithm remains almost constant for a fixed number of processors per shooting patch computation.

Table 4.4 illustrates the variation of the increase in the total number of shooting patch selections for different tolerance values and number of processors. As is seen in this table, the modification introduced for the sake of efficient parallelization increases the total number of shooting patch selections. The percent increase in the total number of shooting patch selections increases with increasing number of processors as is expected. However, for a fixed number of processors, this percent increase decreases with decreasing tolerance values. As is seen in Table 4.4, the percent increase in the number of shooting patch selections remains below 12 % for tolerance values % 60 and smaller for P=128 processors. Hence, this parallelization scheme is recommended for relatively smaller number of processors and tolerance values.

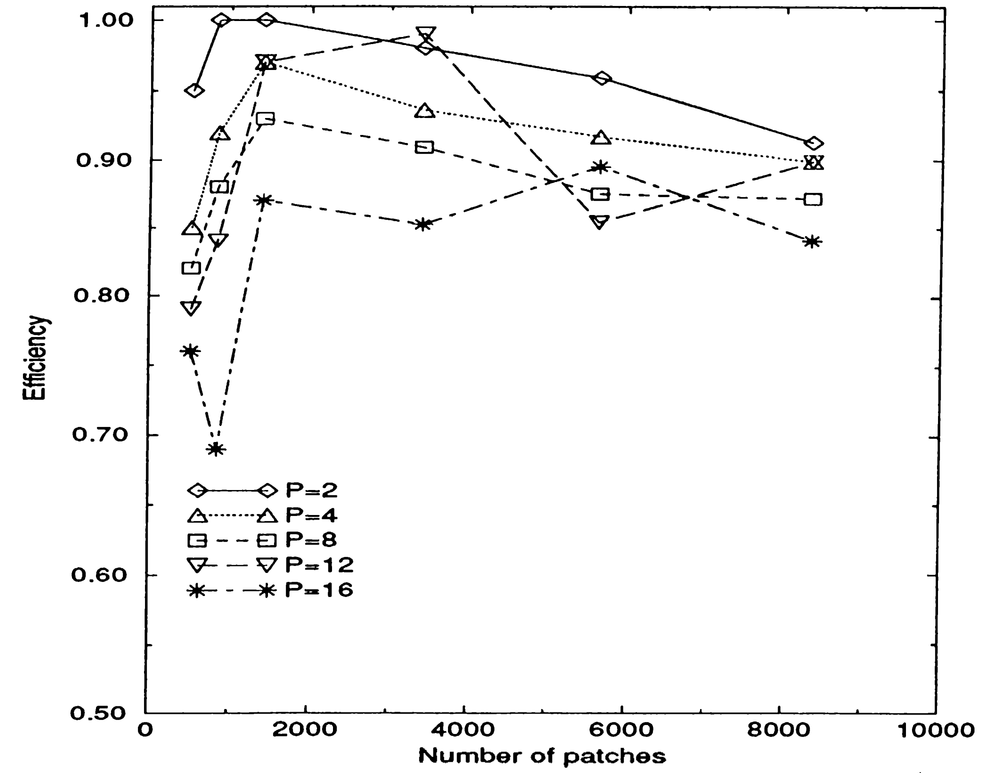


Figure 4.15. Overall efficiency of the Patch Circulation Algorithm

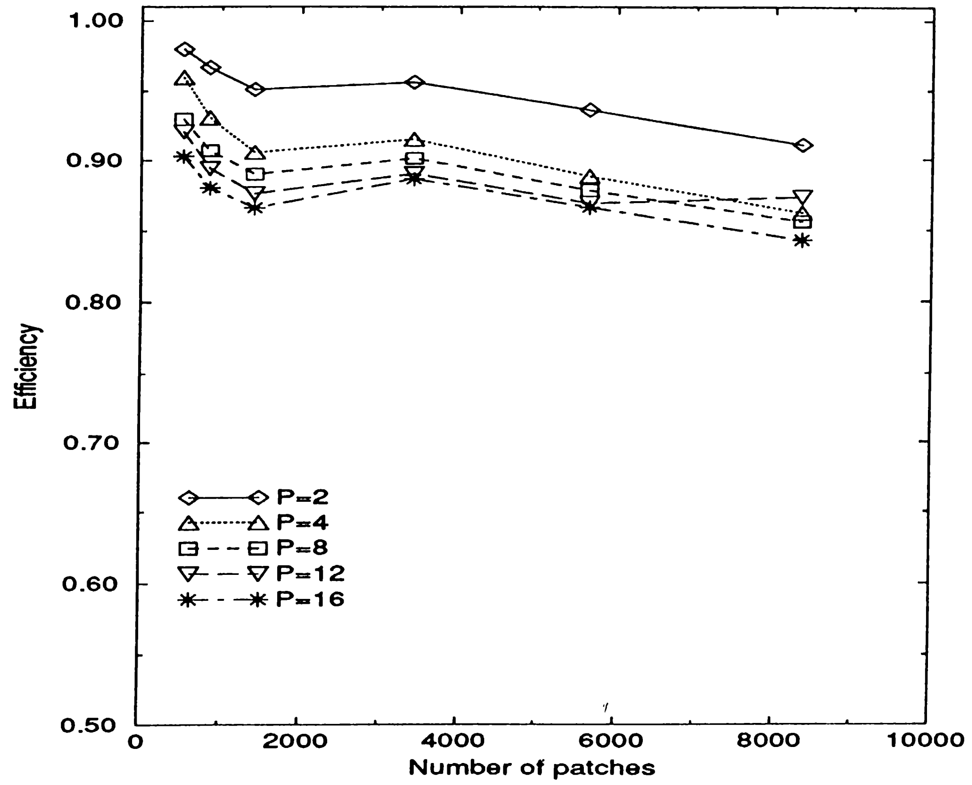


Figure 4.16. Efficiency of the Patch Circulation Scheme per shooting patch

Table 4.4. Total number of shooting patch selections of the parallel algorithm normalized with respect to the sequential algorithm

	Final delta radiosity percentage							
P	75	70	65	60	55	50	45	40
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.93	0.99	1.00	1.00	1.00	1.00	1.00	1.00
4	0.80	1.01	1.00	1.00	1.00	1.00	1.00	1.00
8	1.00	1.00	1.01	1.00	1.00	1.00	1.01	1.00
16	1.07	1.20	1.11	1.03	1.01	1.01	1.01	1.01
32	2.13	1.30	1.25	1.04	1.02	1.02	1.03	1.02
64	4.27	1.50	1.23	1.12	1.06	1.05	1.05	1.04
128	8.53	1.95	1.29	1.12	1.11	1.11	1.08	1.07

4.4 Conclusion

In this chapter, an efficient synchronous parallel progressive radiosity algorithm based on patch data circulation is proposed and discussed. This scheme exploits the Level 1b parallelism mentioned in Chapter 3. The complexity analysis shows that using simple interconnection topologies (as ring) instead of rich topologies (as hypercube) does not degrade the efficiency of the parallel algorithm. The synchronous parallelism is proposed in order to obtain better coherence hence increasing the convergence rate. The proposed parallel algorithm yields good performance for relatively smaller number of processors and tolerance values, as is expected.

For future work, ways of decomposition of the input scene should be investigated. Although scattered decomposition achieves better load balance, better decompositions should be investigated. Furthermore in Phase 2, redundant data may visit the processors. That is, the processors receive the data of the patches which are not visible by their shooting patches. Thus, the volume of communication can be decreased if the exchange of redundant data is avoided.

Chapter 5

Parallelization: Hemicube Merging

This chapter presents the second scheme for parallelization, based on one shooting patch at a time, corresponding to Level 2 of the parallelism level classification for radiosity. The *Hemicube Merging Scheme* again exploits synchronous parallelism with static task assignment. This new scheme is based on a *divide-and-conquer* approach. An efficient communication scheme is proposed for hypercube-connected and ring-connected multicomputers, which decreases the total volume of communications by an asymptotical factor. The proposed parallel algorithms are implemented on iPSC/2 hypercube multicomputer and tested for ring and hypercube-interconnection topology.

5.1 Preliminaries and Data Structures

In what follows, P denotes the number of the processors, $d = log_2^P$ the dimension of the hypercube, N the number of the patches in the environment, R the half-width resolution of the hemicube, $H = 12R^2$ the size of the total hemicube.

The input patches are assumed to be polygons, and have the following type:

```
/* Patch data type */
PatchType = type
  record of
    integer patch_id;
    RayType normal;
```

```
        (x,y,z) vertices[];
        (r,g,b) reflectivity;
        (r,g,b) radiosity;
        (r,g,b) delta_radiosity;
        float   area;
   end;

   {The following declares local patches of the processor }
   PatchList : array [1..N/P] of PatchType;
```

The type definition for the hemicube is as follows:

```
   /* Type definition of the single pixel on the hemicube */
   ItemBufferEntryType = type
      record of
        /* Id of the patch visible at the pixel */
        integer patch_id;
        /* Nearest distance to the source patch on the pixel */
        float   z;
   end;

   /* Type definition of the whole hemicube */
   HemicubeType = type
     record of
       /* top face of the hemicube */
       ItemBufferEntryType +y_buffer[2R][2R];
       /* side faces of the hemicube  */
       ItemBufferEntryType +x_buffer[2R][R],
                           -x_buffer[2R][R],
                           +z_buffer[2R][R],
                           -z_buffer[2R][R];
   end;
```

Thus, the hemicube is abstracted by arrays of static size (one of size $2R\times 2R$, four of size $2R\times R$, one array for each face). Recall that messages in iPSC/2 are sent from and received into contiguous memory buffers in the local memories of the processors. Hence, data to be communicated must be stored in contiguous memory locations. Thus, the hemicube is represented as a linear array of contiguous entries of type **ItemBufferEntryType** (See Figure 5.1). With the help of the casting facility of the C Programming Language, it is possible to visualize the hemicube first as a record in item buffer filling step, and as an array thereafter. Being able to manipulate the hemicube as a linear array gives the facility of subdividing the hemicube independent of its partitioning into faces. In some of the algorithms presented, this capability is used to subdivide the final hemicube evenly among the processors (Figure 5.1).

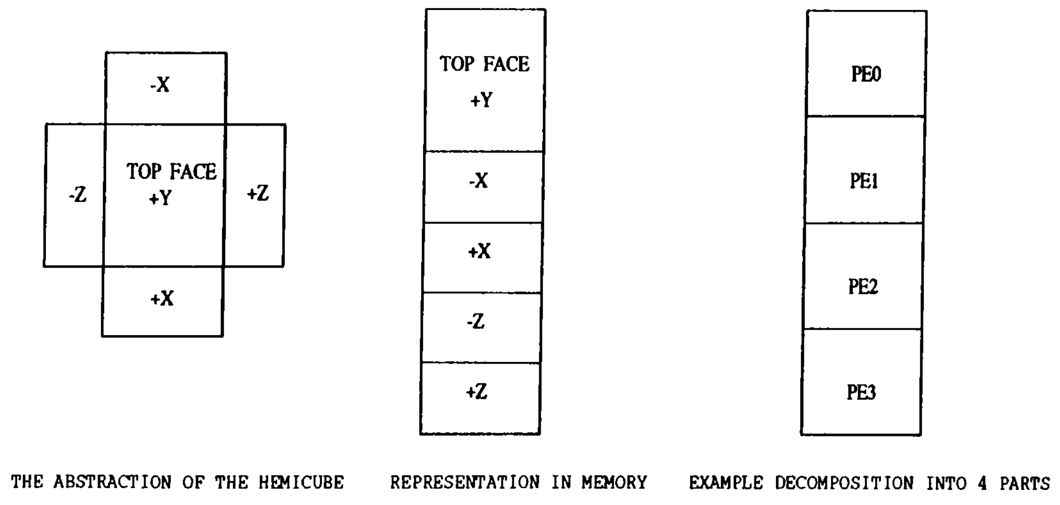


Figure 5.1. Abstraction and Representation of the Hemicube

In the algorithms presented in this chapter, a single hemicube is decomposed among the processors and each processor performs the computations for its part of the hemicube. Thus, we have the subprogram **FormFFVector()** in Figure 5.2, which computes a form-factor vector for the input range of the hemicube by scanning the input portion and adding the delta form-factors of the pixels of the same patch to compute the correct form-factor value for that patch.

Note that the subprogram visualizes the hemicube as a linear array. The subprogram is called by specifying the start and end index of the linear array of the pixels on the hemicube.

```
subprogram FormFFVector
begin
      Initialize form_factors[] vector entries to 0.0;
      for i=start_index to end_index do
         patch_index = hemicube[i].patch_id;
         form_factors[patch_index] = form_factors[patch_index] +
                                     delta_form_factors[i];
      endfor
end subprogram
```

Figure 5.2. Algorithm for Form-Factor Vector Construction

5.2 Parallelization

The Hemicube Merging Scheme is based on a *divide-and-conquer* approach. In this scheme, a single shooting patch is selected (similar to the sequential algorithm), and each processor fills its local hemicube by its local patch subset of the global patch data, as if the environment consists only of its local patches, by executing the sequential item fill algorithm with its local patches. Then, in the later phase, the processors merge their hemicubes to compute the final hemicube using the distance values at the pixels. The detailed algorithm for Hemicube Merging Scheme is illustrated in Figure 5.3.

The following subsections describe the steps of the Hemicube Merging Scheme in detail.

5.2.1 Step 1: Shooting Patch Selection

In this step, the single shooting patch with the maximum unshot energy ($\Delta B_i A_i$) among all patches distributed in the local memories of the processors, is selected. This step consists of two phases: First, each processor selects the patch i with maximum $\Delta B_i A_i$ among its local patches; then in the communication phase, these P patches are compared and the maximum energy holding patch is selected as the shooting patch.

Note that this problem is very similar to Phase 1 of the Patch Circulation

At each iteration :

Step 1: The shooting patch, which has the globally maximum $\Delta B_i A_i$, is selected.

Step 2: Each processor projects its local patch data onto its local hemicube by executing the sequential algorithm of item buffer fill. As a result, P different hemicubes, each filled with local patches of another processor, are produced.

Step 3: GMERGE (Global MERGE step) In this step, P hemicubes produced at the end of Step 2 are merged, based on the distance of patches at hemicube item buffer pixels, to compute the final hemicube corresponding to the shooting patch.

Step 4: Each processor is given a different H/P part of the final hemicube, and each processor computes a form-factor vector using its part. This operation uses the facility of subdividing the hemicube independent of its partitioning to its faces (Figure 5.1), and subroutine **FormFFVector** discussed in the previous section.

Step 5: The P form-factor vectors produced at the end of Step 4 are added to compute the final form-factor vector corresponding to the shooting patch.

Step 6: Having computed the form-factors, each processor adds the contributions from the shooting patch to its local patches using the form-factor entries corresponding to its local patches.

Figure 5.3. Algorithm for Hemicube Merging Scheme

Scheme, discussed in Section 4.2.1, with a slight difference. In Phase 1 of the previous scheme, the processors communicate and compare P shooting patches, and as a result, P patches are selected as the simultaneously shooting patches. However, in Step 1 of the Hemicube Merging Scheme, only one patch is selected as the shooting patch, therefore the communicated buffers have the size of only one patch data.

Thus, the algorithm for Step 1 of the Hemicube Merging Scheme is very similar to the algorithm proposed for Phase 1 of the Patch Data Circulation Scheme shown in Figures 4.1 and 4.3 for the ring and hypercube topologies, respectively. Note that, in this case, the array *PartialMaximums* has the size of 1, instead of P as shown in these figures.

5.2.2 Step 2: Hemicube Production Step

In this step, each processor projects and fills its local patches onto its local hemicube. This step does not require interprocessor communication and is the sequential algorithm of hemicube fill with the processor's local patches. As discussed in Section 2.2.2, the patches are passed through a projection pipeline consisting of *i*)viewing transformation, *ii*) clipping, *iii*) perspective projection, *iv*) scan conversion of the local patches onto each face of the hemicube [14].

The crucial factor that affects the performance of this phase is the computational load balance among the processors. In the following step (Step 3), the processors are synchronized, therefore the computational load in this step should be distributed evenly in order to minimize the processors' idle time, thus increasing the performance of Step 2. As discussed in Section 4.2.2, the complexity of projection of an individual patch onto the hemicube faces depends on several geometric factors. Chapter 4 introduces two types of decompositions: *tiled* and *scattered* decompositions (as illustrated in Figure 4.9). Recall that, it is expected that neighbour patches require almost equal amount of computation in Step 2 (e.g. in visibility test stage, neighbour patches may all be invisible by the shooting patch; or in the scan-conversion stage, neighbour patches may project onto almost equal number of hemicube pixels, hence require almost equal amount of computation). In scattered decomposition, neighbour patches are distributed to different processors, so the patches that belong to the same object are evenly shared by the processors. Hence, scattered decomposition is expected to achieve better load balance in this step.

So, assuming a perfect load balance in Step 2, the complexity of the hemicube production process is:

$$T_{step2} = \frac{N}{P} \times T_{PROJ} \tag{5.1}$$

where T_{PROJ} is the average time to project one patch onto the hemicube.

5.2.3 Step 3: Hemicube Merge Step

At the end of Step 2, each processor has constructed a hemicube filled with its local patches; hence P distinct hemicubes have been formed that correspond to hemicubes handling the occlusions and the visibility among the local patches of the processors only. In Step 3, these P hemicubes are *merged* to a single hemicube, which corresponds to the correct hemicube for the shooting patch, that solves the occlusions and visibility between the shooting patch and all other patches in the input scene.

The merging process is as follows. For each hemicube pixel, the patch with the nearest distance (among all the P patches which are visible at that pixel stored at the P hemicubes), is selected as the visible patch for that pixel on the final hemicube.

In the following subsections; first, the naive algorithm for the hemicube merging process is described, and then the new algorithms which decrease the total volume of concurrent communication by an asymptotical factor are presented for the ring and hypercube topologies.

Naive Merge Algorithm for Ring Topology

The naive algorithm for merging the P hemicubes on the ring topology is shown in Figure 5.4. The hemicube merging for the ring topology is performed in $P-1$ concurrent communications as follows: The processors start with their produced hemicubes stored in their local memories initially. Then, at each iteration of the algorithm, the processors send their partial result to the next processor in the ring, and receive a new partial result from the previous processor in the ring. Thus, effectively the partially merged hemicubes are circulated as each processor *merges* its original local hemicube with cumulatively merged other hemicubes. Note that, the original local hemicubes of the processors remain unchanged during the circulation of the partially merged hemicubes.

```
Ring_MergeHemicubes1
begin
      /* Initially processor has its local hemicube in */
      /* linear array Hemicube[] in its local memory.  */
      nextnode = (mynode + 1) mod no_processors;
      copy Hemicube[] to PartialHemicube[];

      for i=1 to P-1 do
          send PartialHemicube[] to nextnode;
          receive PartialHemicube[];
          for j=0 to H-1 do
              if PartialHemicube[j].z > Hemicube[j].z then
                 PartialHemicube[j].patch_id = Hemicube[j].patch_id;
                 PartialHemicube[j].z = Hemicube[j].z;
              endif
          endfor
      endfor
end
```

Figure 5.4. Ring Algorithm for Naive Hemicube Merging

At the end of $P-1$ concurrent exchanges, each processor has a copy of the correct hemicube corresponding to the shooting patch in its local variable *PartialHemicube*. The complexity of this operation for the ring topology is:

$$T_{HCM1} = (P-1) \times t_{SU} + (P-1) \times H \times T_{TR} + (P-1) \times H \times T_{MERGE} \quad (5.2)$$

where H is the size of the hemicube in pixels, T_{TR} is the time spent to exchange one pixel, T_{MERGE} is the time to perform the merging operation for a single pixel.

Naive Merge Algorithm for Hypercube Topology

The naive algorithm for merging the P hemicubes on the SIMD hypercube topology succeeds to decrease the complexity of Step 3 from $O(P)$ to $O(log_2 P)$. The new algorithm for the SIMD hypercube is illustrated in Figure 5.5. At each iteration of the d iterations, the processors exchange their current hemicube with a different neighbour processor, and merge these hemicubes. Note that, the algorithm shown in Fig. 5.5 uses the communication protocol illustrated in Figure 3.3. In fact, for merging the hemicubes the algorithm performs the *GMIN* (Global Minimum) operation on the linear arrays of hemicube pixels with respect to the patch distances at the pixels.

```
Hypercube_MergeHemicubes1
begin
    /* Initially processor has its local hemicube in */
    /* linear array Hemicube[] in its local memory.  */
    for i=0 to d-1 do
        dnode = mynode ⊕ $2^i$;
        send Hemicube to dnode;
        receive ItsHemicube from dnode;
        for j=0 to H-1 do
            if ItsHemicube[j].z < Hemicube[j].z then
                Hemicube[j].patch_id = ItsHemicube[j].patch_id;
                Hemicube[j].z = ItsHemicube[j].z;
            endif
        endfor
    endfor
end
```

Figure 5.5. Hypercube Algorithm for Naive Hemicube Merging

At the end of d concurrent exchanges, each processor has a copy of the correct hemicube corresponding to the shooting patch in its local variable *Hemicube*. The complexity of this operation for the hypercube topology decreases to:

$$T_{HCM1} = d \times t_{SU} + d \times H \times T_{TR} + d \times H \times T_{MERGE} \tag{5.3}$$

d is the dimension of the hypercube.

So, the naive algorithm for the hypercube topology requires asymptotically less number of communications, volume of communications and amount of computation than the naive ring topology. However, the hypercube algorithm still requires excessive amount of computation and communication for merging the hemicubes, and the extra communications and computations become more crucial with the increasing number of processors. The following subsection proposes efficient hemicube merging algorithms for ring and hypercube topologies.

Efficient Merge Algorithm for Ring Topology

First, note that after Hemicube Merging Step (Step 3), in Step 4, that is the form-factor vector computation step, each processor is given the task of scanning its H/P part of the final hemicube and constructing a form-factor vector for that section of the hemicube only. Therefore, there is no need to duplicate the whole final hemicube in all the processors' local memory, thus the algorithms given in Figs. 5.5 and 5.4 involve both communication and computation redundancy for the merging operation.

Second, recall that the merging operation over the hemicube pixels is in fact a *Global MINimum* operation with respect to the patch distances. The minimum function is

- **Commutative:** $Value_1$ MIN $Value_2$ = $Value_2$ MIN $Value_1$.
- **Associative:**
  $Value_1$ MIN ($Value_2$ MIN $Value_3$) = ($Value_1$ MIN $Value_2$) MIN $Value_3$.

That is, the order of testing the input values does not change the final result of the MIN function.

The contribution computations in Phase 4 of the *Patch Circulation Scheme* as discussed in Section (4.2.4) also have the same commutativity and associativity properties. That is, the order of adding the contributions from more than one shooting patches does not change the final total contributions from those patches if the inter-contributions among the shooting patches are ignored, which was the case for Chapter 4. Hence, the parallel solutions proposed for Phase 4 of the Patch Circulation Scheme can also be used in this step with a slight difference. In the previous algorithms, the contributions from P shooting patches are *summed* and as a result, each processor holds the slice of the global contributions corresponding to its local patches only. In this problem, the hemicubes are *merged* using the MIN function and as a result, each processor holds only the slice of the hemicube which it will process in the next step (Step 4). Figure 4.12 used to illustrate the execution of the circulation scheme can be adapted here, substituting **H** arrays for **U** vectors, and using the MIN function for merging the **H** vectors.

Thus, the time complexity of Step 3 using the efficient communication algorithm on the ring topology is:

$$T_{HCM2_{RING}} = P \times t_{SU} + H \times T_{TR} + H \times T_{MERGE} \tag{5.4}$$

Efficient Merge Algorithm for Hypercube Topology

Observe that, the communication efficient algorithm proposed for the ring topology requires P number of communications. The operation can be performed in d steps without increasing the volume of communication and the computational load, on the hypercube topology.

The communication efficient hypercube algorithm again uses the communication protocol presented in Figure 3.3, but this time exchanging and processing recursively decreasing portions of the hemicube, instead of communicating the whole array. In the new scheme, each processor starts with the whole hemicube portion. Then the algorithm proceeds as follows: At each iteration of the algorithm, each processor sends one half of its current portion to a different neighbour processor and receives that processor's other half of the current portion. So, each processor has two copies of one half of its current portion. Then it merges these two halves into its local half. At the end of the iteration, the range of the current portion is divided by 2 in order to correspond to the merged half. An example execution of the new algorithm for 2D hypercube is presented in Figure 5.6. The detailed algorithm for the new efficient approach is illustrated in Figure 5.7. In the algorithm, H is assumed to be an integer power of 2, but the implementation accepts any value of H.

The amount of computation and volume of communication is proportional to $H/2^{k+1}$ at step k, so the total work is:

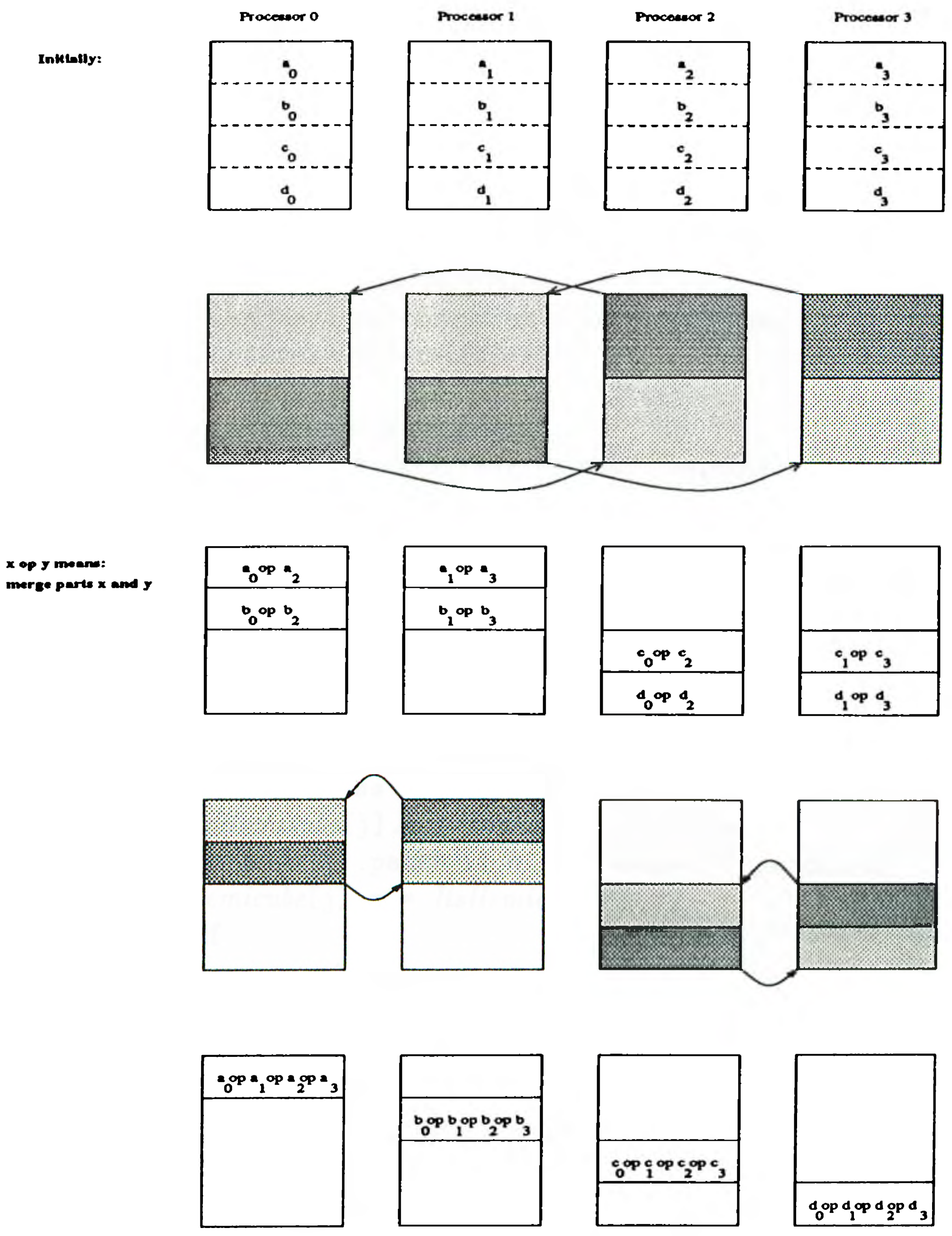


Figure 5.6. Example execution of *MergeHemicubes2* on a Hypercube with 4 Processors

```
Hypercube_MergeHemicubes2
begin
     /* $H$ is assumed to be a power of 2. */

     send_rec_size = H/2;
     send_index = recv_index = 0;
     for i=d-1 downto 0 do

         if $i^{th}$ bit of mynode is 0 then
            send_address = send_index + send_rec_size;
            recv_address = recv_index;
         else
            send_address = send_index;
            recv_address = recv_index + send_rec_size;
         endif

         dnode = mynode $\oplus$ $2^i$;
         send from Hemicube[send_address] with
            size send_rec_size to dnode;
         receive into ItsHemicube[recv_address] with
            size send_rec_size from dnode;

         for j=recv_address to recv_address+send_rec_size do
            if ItsHemicube[j].z < Hemicube[j].z then
               Hemicube[j].patch_id = ItsHemicube[j].patch_id;
               Hemicube[j].z = ItsHemicube[j].z;
            endif
         endfor

         if $i^{th}$ bit of mynode is 1 then
            send_index = send_index + send_rec_size;
            recv_index = recv_index + send_rec_size;
         endif
         send_rec_size = send_rec_size / 2;
     endfor
     end
```

Figure 5.7. Hypercube Algorithm for Communication Efficient Hemicube Merging

$$H/2^1 + H/2^2 + ... + H/2^d = H \times (P-1)/P \approx H$$

Therefore, the complexity of the algorithm is as follows :

$$T_{HCM2_{HYPER}} = d \times T_{SU} + H \times T_{TR} + H \times T_{MERGE}$$

In the algorithm presented in Figure 5.7, the communication is assumed to be synchronous, that is: the processors wait until send and receive operations are completed. However, it is possible to further speed up the algorithm by overlapping communication and computation with the use of asynchronous communication. In *asynchronous send* command, the processor continues to the next instruction, where a hardware router in the processor copies the addressed data to the bidirectional link concurrently. In *asynchronous receive* command, if the message has arrived, the message data is copied from the system buffer to memory address pointed; otherwise the processor continues to the next instruction and the received messages thereafter are received into the system message buffers. Also there is a utility **msgwait** to test if the message has been received by the processor and is ready in the system buffer.

Then, the communication efficient algorithm proceeds as follows: The processor issues an asynchronous receive command in the beginning, and while the processor does the computations for step i, it issues another asynchronous receive for step $i+1$ concurrently.

Note that, in Figure 5.7, the algorithm uses an additional hemicube array $ItsHemicube$ for simple discussion. In fact, the synchronous communication protocol can be performed without any extra storage. When the processor sends the corresponding half of the current portion of the hemicube, that portion will not be used thereafter. Then, the processors can receive the into the sent portion of the local hemicube array. However, in asynchronous communication, it is possible that the new partial results for iteration $i+1$ has arrived while the processor does the computations for iteration i, overwriting the unprocessed entries. Therefore, extra storage of size $H/2$ (the maximum hemicube half portion size that will be used at iteration 1) is required for asynchronous communication.

Performance Analysis of Step 3

There are two crucial factors that affect the performance in this step: load balance and volume of communication. Perfect load balance is achieved in this phase if almost equal number of hemicube pixels are distributed to each processor at the end of Step 3, assuming that merging operation takes constant amount of time for each pixel. Volume of communication is reduced asymptotically by using the proposed scheme for communication for the ring and SIMD hypercube topologies. At the end of this chapter, a further modification to *Hemicube Merging Scheme* in order to decrease the volume of communication further for MIMD hypercube-connected multicomputers, is presented.

5.2.4 Step 4: Form-Factor Vector Construction Step

At the end of Step 3, each processor has received a different H/P portion of the final hemicube correctly. The correct portions of the linear hemicube array at the processors' local memories are:

```
Processor  0: [0  ..(H/P-1)]
Processor  1: [H/P..(2H/P-1)]
.
.
.
Processor  i: [i*H/P..(i+1)*H/P-1]
.
.
.
Processor P-1: [(P-1)H/P..(H-1)]
```

In Step 4, the processors construct a form-factor vector for their local portion of the final hemicube. This process makes use of the algorithm **FormFFVector** described in the preliminaries section of this chapter. The delta form-factors are stored in a look-up table in order to prevent recalculation for each shooting patch.

Note that, the patches that are visible at the pixels of the local hemicube portion for a particular processor are not necessarily the local patches of the hemicube. As a result of this step, P processors form a total of P form-factor vectors corresponding to P different portions of the final hemicube.

5.2.5 Step 5: Form-Factor Vector Addition

At the end of Step 4, each processor has computed a form-factor vector from the shooting patch corresponding its portion of the hemicube. In this step, these vectors must be added to form the final form-factor vector for the source patch corresponding to the whole hemicube.

Similar to Step 3, the vector sum operation required for this step is commutative and associative, and at the end of this step, the processors need only those entries of the form-factor vector corresponding to their local patches to compute the contributions to their local patches from the shooting patch. Hence, each processor needs the slice of the final form-factor vector corresponding to its local patches only. Therefore the communication efficient schemes proposed for Step 3 can also be used in this step. However in this step, the operation applied to the linear array entries is *summation*, instead of merging the entries as in Step 3.

So, at the end of Step 5, each processor holds in its local memory:

```
Processor 0 has computed FF[0..N/P-1],
Processor 1   "     "     FF[N/P..2N/P-1]
  .
  .
  .
Processor i   "     "     FF[i*N/P..(i+1)*N/P-1]
  .
  .
  .
Processor P-1 "     "     FF[(P-1)N/P..N-1] correctly.
```

5.2.6 Step 6: Contribution Computation

Having received the form-factor vector slice from the shooting patch to its local patches, each processor computes the contributions to its local patches using Eq. (2.17) and adds this contribution to their radiosity and delta radiosity values.

5.3 An Improvement: Hemicube Division Scheme

Now we present the *Hemicube Division Scheme* for the MIMD hypercube-connected multicomputers which decreases memory requirements for storing the hemicubes at local memories of the processors, while improving the parallelism of *Hemicube Merging* phase previously discussed. The idea, is to distribute the four hemicube side faces to four subcubes of the hypercube, each subcube filling a distinct side face with all the patches, and then all the processors filling the top face in the later phase. It is not an exception that the number of patches that are visible through different hemicube faces is not equal. This results in load imbalance because the amount of work is proportional to the number of patches visible. In order to solve this problem, the second phase (top face filling) is executed using a master-slave scheme.

5.3.1 Face Allocation to Subcubes

There are two possibilities for allocating the 4 subcubes for the hemicube's four side faces:

- **homogenous sized subcubes:** Assuming each side face will have nearly the same number of patches projected onto it, the d-dimensional hypercube is partitioned into 4 (d-2)-dimensional subcubes. Figure shows an example homogeneous allocation for 4-dimensional hypercube.
- **heterogeneous sized subcubes:** It is possible that one or more of the side faces have a large number of patches projected onto them compared to the other faces. If homogeneous sized subcubes are used in these situations, it is possible that the subcubes that have small number of patches projected process their side face, and then the top face; and still

the subcube that processes the face with maximum number of patches continues projection of the patches onto its face. To solve this problem, the dimension of the subcube for the face with maximum number of patches, has to be increased to reduce the time of execution for this face. In heterogeneous type of allocation, this face will be processed by a (d-1)-dimensional subcube, the next by a (d-2)-dimensional subcube, and the lower two by (d-3)-dimensional subcubes. Figure 5.8 shows an allocation of this kind on 4-dimensional hypercube.

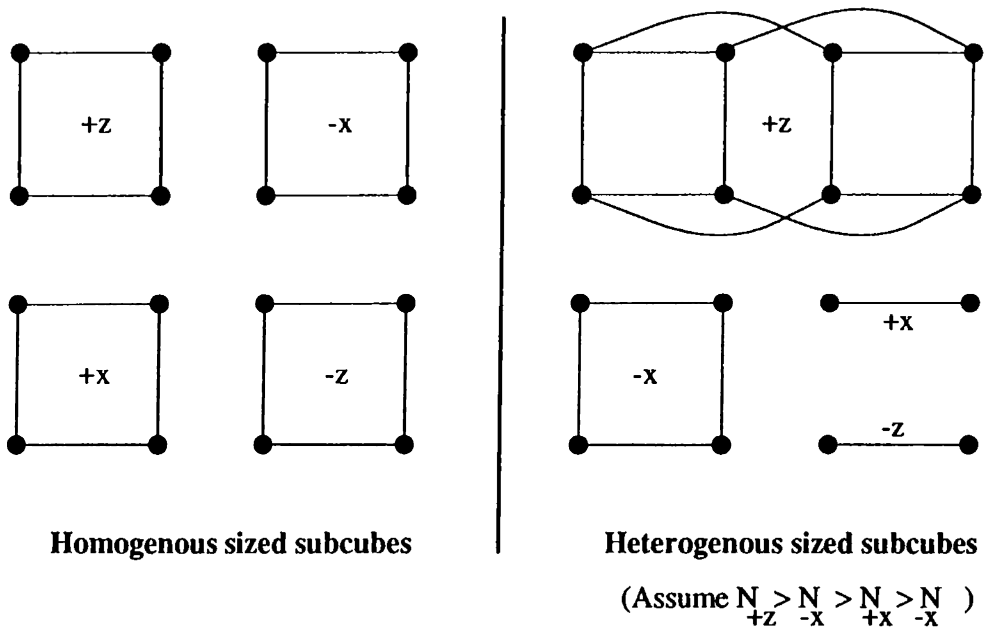

+z
-x
+x
-z
Homogenous sized subcubes
+z
-x
+x
-z
Heterogenous sized subcubes
(Assume $N_{+z} > N_{-x} > N_{+x} > N_{-x}$)

Figure 5.8. Subcube Allocation for Hemicube Faces

5.3.2 Data Distribution

In order for a subcube to fill its side face, it should have all the patches in the environment. However, the original data distribution discussed before can allow only $N/4$ patches for a subcube in *homogenous sized subcubes* allocation scheme. So, a limited amount of space redundancy is used.

In the new distribution, the processors store only color information of their N/P local patches. In addition to this, the geometric information (patch vertices, normal vector) of the local patches of the 4-processor set is stored at each processor. Figure 5.9 illustrates the data distribution for 4-dimensional hypercube. Note that the size of the geometric data is $4N/P$ and of the color data is N/P. If N is too large to store the geometric data with $4N/P$ entries, then only N/P geometric data is stored for the processors' local patches, and these data are circulated in a ring of size 4. Note that the ring size is constant and independent of the cube size.

5.3.3 Hemicube Division Scheme Algorithm

Each iteration of the algorithm consists of 4 phases: preprocessing phase, side face filling phase, top face filling (master-slave) phase, contribution addition phase.

- **i)Preprocessing phase** In this phase, each processor computes, for its local N/P patches, the number of patches falling onto each face of the hemicube, and at the same time prepares a mask of 5 Booleans, one for each hemicube face. Then, a GSUM among all the processors is applied to find the total number of patches visible through the faces, and GCOL operation is applied among the 4-processor group such that each processor holds the visibility information for all its patch geometry data. The computation of the number of local patches onto the hemicube faces is performed using an estimation based on simple tests whether the patch vertices fall onto the hemicube faces. This is accomplished by bringing the destination patches into the viewing coordinate system by multiplying the patch vertices with the viewing matrix and executing simple comparisons of the projected destination patch vertex coordinates.

- **ii)Side face filling phase** Depending on the side face counters computed in preprocessing phase, a choice is made between *homogeneous* or *heterogeneous sized subcubes* schemes.

  $S_1 = max\{N_{+z}/2^{d-2}, N_{-z}/2^{d-2}, N_{+x}/2^{d-2}, N_{-x}/2^{d-2}\}$

  sort $N_{+z}, N_{-z}, N_{+x}, N_{-x} \rightarrow \{N_1, N_2, N_3, N_4\}$ s.t. $N_1 \geq N_2 \geq N_3 \geq N_4$

  $S_2 = max\{N_1/2^{d-1}, N_2/2^{d-2}, N_3/2^{d-3}, N_4/2^{d-3}\}$

  S_1 and S_2 are the maximum of the execution times for the subcubes in different face allocation schemes, that is the time to complete the projection. If $S_1 \leq S_2$ then the *homogeneous sized subcubes*, else the *heterogeneous sized subcubes* scheme is employed. Depending on the scheme selected, each subcube is allocated a side face and executes Steps 2, 3 and 4 of the *hemicube merging* scheme with the dimension of that subcube and only one face of the whole hemicube. Having finished filling with all the patches, the processors within a subcube merge their side face and form a form-factor vector from their side face as described above.

- **iii) Top face filling (master-slave) phase** When a subcube has completed filling its side face, all its processors are idle. So, they will start filling the top face. This is done using a *master-slave* approach among the 4-processor groups. The processors [0..P/4-1] are selected as the masters for different 4-processor groups. When a slave processor's request comes, the master sends the start index of the next available patch in geometric data duplicated in its 4-processor group. Then, the slave processor fills the top face with the packet of patches starting at this index. The packet length, which is determined in the beginning of the program, is proportional to the total number of patches. When all the $4N/P$ patches are scan-converted for the top face for all the subcubes, a GMERGE operation of the top face is performed among all the processors, then the form-factors computed from this phase is added to the form-factor vector computed.
- **iv)Form Factor and Contribution Addition Phase** The distinct form-factor vectors produced at the end of phase (iii) are added and each processor computes the contributions from the shooting patch to its local patches.

5.3.4 Performance Analysis of Hemicube Division Scheme

Recall that the complexity of the hemicube merging operation of the Hemicube Merging Scheme is:

$$T_{HCM} = d \times t_{SU} + H \times T_{TR} + H \times T_{MERGE} \tag{5.5}$$

Assuming that the perfect load balance is achieved for the Homogeneous Subcubes Allocation for the Hemicube Division Scheme, and ignoring the communication caused by the master slave messages, the complexity of hemicube merging phase of the Hemicube Division Scheme can be written as:

$$T_{HCD} = T_{HCD_{side}} + T_{HCD_{top}} \tag{5.6}$$

where $T_{HCD_{side}}$ is the time required for the subcube to merge the allocated side face, $T_{HCD_{top}}$ is the time required by all the processors to merge the hemicube top face.

Similar to Eq. 5.5, the complexity for the side and top faces can be written as:

$$\begin{aligned} T_{HCD_{side}} &= (d-2)t_{SU} + \frac{H}{6}T_{COMP} && (5.7) \\ &\approx \frac{H}{6}T_{COMP} && (5.8) \\ T_{HCD_{top}} &= dt_{SU} + \frac{H}{3}T_{COMP} && (5.9) \\ &\approx \frac{H}{3}T_{COMP} && (5.10) \end{aligned}$$

where $T_{COMP} = T_{TR} + T_{MERGE}$. Then, T_{HCD} can be rewritten as:

$$\begin{aligned} T_{HCD} &= \left[\frac{1}{6} + \frac{1}{3}\right] HT_{COMP} && (5.11) \\ &= \frac{H}{2}T_{TR} + \frac{H}{2}T_{MERGE} && (5.12) \end{aligned}$$

Thus, the hemicube division scheme decreases the total volume of communication and the hemicube merging operation by a factor of 2.

5.4 Results

The hemicube merging scheme has been implemented on the Intel's iPSC/2 multicomputer. The proposed parallel algorithms are experimented for six different scenes with 522, 856, 1412, 3424, 5648, 13696 patches. The test scenes are selected as house interiors consisting of objects such as chairs, tables, windows, lights in order to represent realistic 3D environments.

Table 5.1 illustrates the effect of the decomposition scheme on the performance of the local hemicube production step (Step 2) of the parallel algorithm. Parallel timings in this table denote the average time in seconds for execution of Step 2 for a shooting patch. The parallel timings denote the quality of the decomposition schemes. Note that, as the number of processors increases, the load balance quality of the scattered decomposition increases in comparison with that of the tiled decomposition.

Table 5.1. Effect of the Patch Data Decomposition Type on Performance of Hemicube Production Step (Step 2)

N	P	Tiled Decomposition	Scattered Decomposition	Percent Decrease
522	4	1.403	1.041	25.80
	8	0.833	0.561	32.65
	16	0.538	0.315	41.45
856	4	1.969	1.341	31.89
	8	1.110	0.731	34.14
	16	0.739	0.418	43.44
3424	4	4.802	3.064	36.19
	8	2.694	1.519	43.62
	16	1.583	0.803	49.27
5648	4	6.935	4.445	35.90
	8	4.020	2.217	44.85
	16	2.392	1.166	51.25
8352	4	9.249	6.526	29.44
	8	5.326	3.326	37.55
	16	3.108	1.711	44.95

Table 5.2 shows the effect of the new efficient communication scheme on the performance of Step 3. As is seen in Table 5.2, the time required by the naive communication algorithm increases proportionally to the dimension d of the hypercube, while the proposed efficient communication scheme requires almost equal amount of time with increasing dimension d of the hypercube. As is seen in this table, the advantage of the efficient scheme increases substantially with the increasing dimension of the hypercube.

Figure 5.11 illustrates the overall efficiency curves for the Hemicube Merging Scheme. In Fig.5.11, the efficiency curves are constructed using

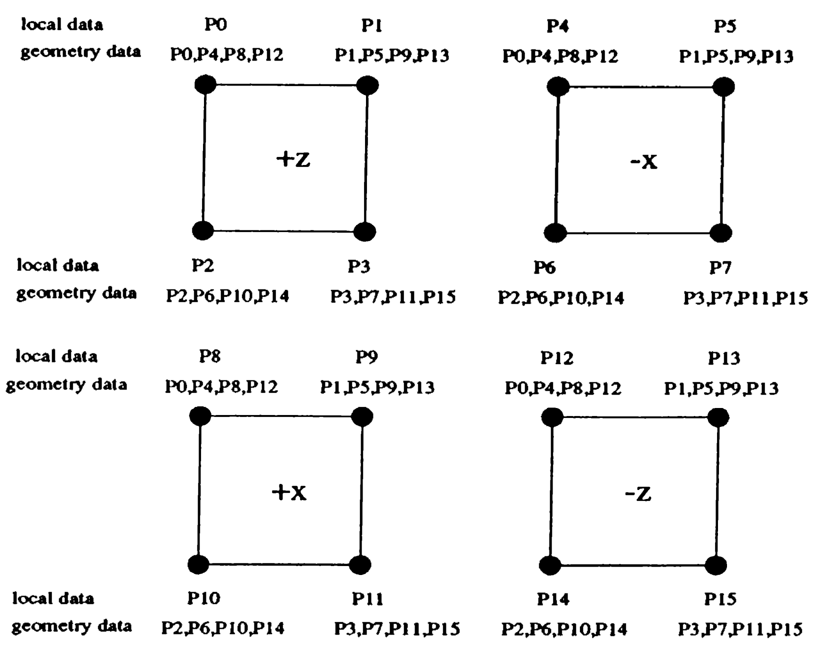
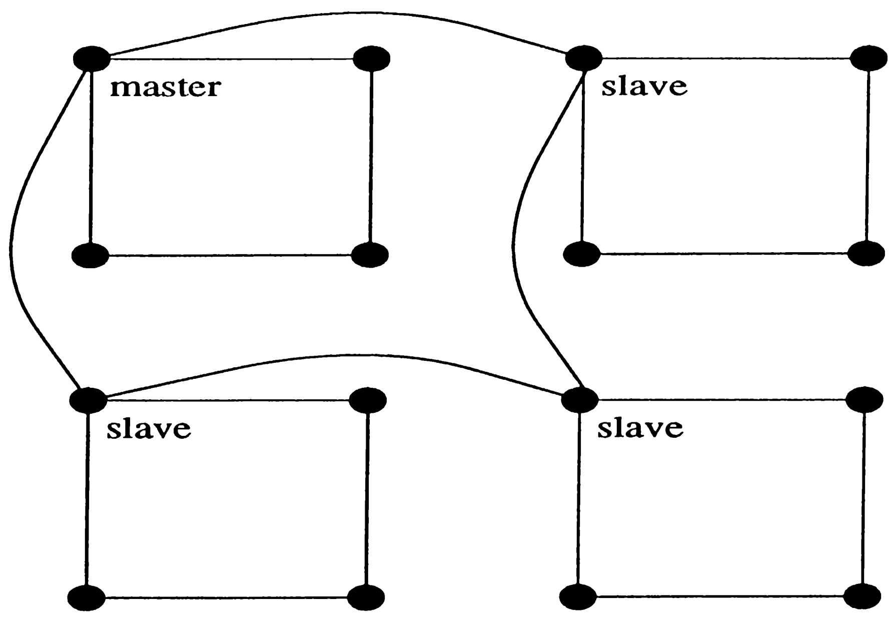


Figure 5.9. Distribution of the Geometry Data

Table 5.2. Performance of the proposed parallel hemicube merge algorithm on Hypercube Topology

R	d	Naive Communication	Tightly-Coupled		Loosely-Coupled	
			time	speedup	time	speedup
50	1	460	283	1.63	276	1.67
	2	1017	448	2.27	439	2.32
	3	1599	543	2.94	523	3.06
	4	2190	592	3.70	576	3.80
75	1	962	633	1.52	617	1.56
	2	2160	1005	2.15	984	2.20
	3	3441	1214	2.83	1177	2.92
	4	4680	1319	3.55	1290	3.63



The 4-Processors in Master-slave scheme

Figure 5.10. Master-Slave scheme

$$Efficiency = \frac{1}{P}\frac{T_{SEQ}}{T_{PAR}}$$

Here, T_{SEQ} and T_{PAR} denote the execution time taken for the sequential and the parallel algorithm on P processors, respectively to converge to the same tolerance value. Note that, the total shooting patches required for convergence of the parallel algorithm is the same for that of the sequential algorithm. As is seen in Fig.5.11, the efficiency values for a particular number of processors increases with increasing number of patches N because of the almost constant amount of communication required since the hemicube communication is the most time-consuming part. Thus, the granularity increases with increasing number of patches in the scene.

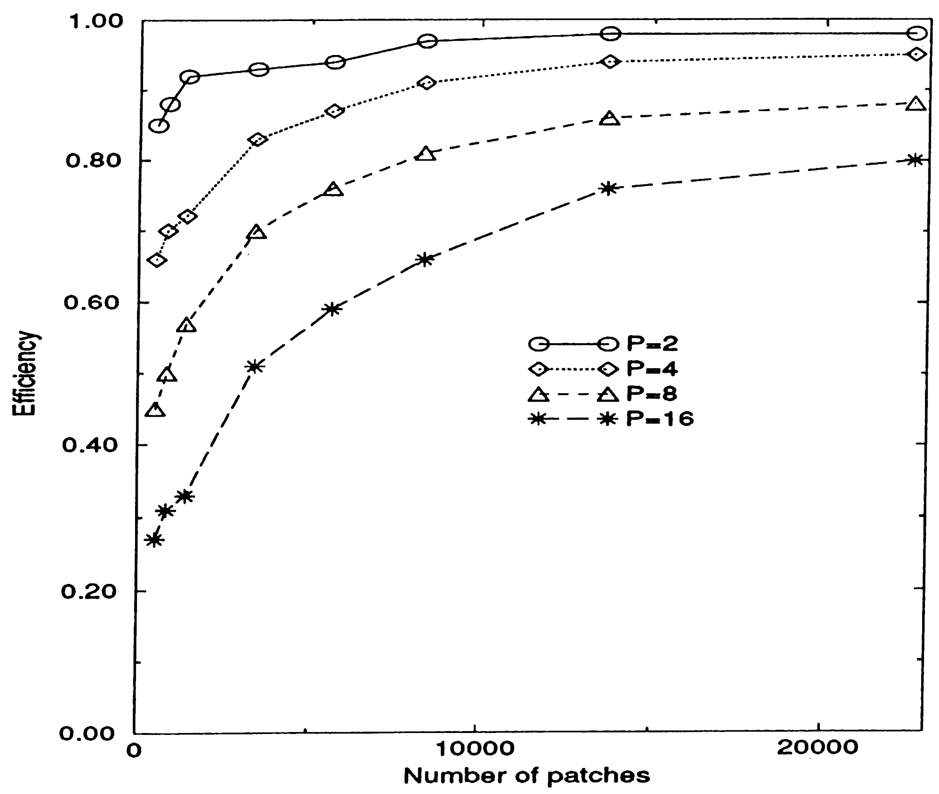


Figure 5.11. Efficiency of the Hemicube Merging Scheme

Table 5.3 shows the execution times of the distributed form-factor computation during a single iteration of the parallel algorithm. Note that, the advantage of the Hemicube Division Scheme decreases with increasing number of patches. This result is due to the fact that the load imbalance among different side faces of the hemicube becomes more crucial with increasing number of patches. Furthermore, the performance of the Hemicube Division Scheme increases in comparison to the Hemicube Merging Scheme with increasing number of processors.

Table 5.3. Effect of the Hemicube Division Scheme on the Performance of the Parallel Solution

N	P	Hemicube Merging Scheme (sec)	Hemicube Division Scheme (sec)	percent decrease
522	4	1.632	1.449	11.21
	8	1.196	0.934	21.90
	16	0.981	0.689	29.77
856	4	1.945	1.815	6.68
	8	1.374	1.109	19.29
	16	1.091	0.788	27.77
3424	4	3.803	3.328	1.25
	8	2.250	2.151	4.40
	16	1.545	1.414	8.48
8352	4	7.508	8.573	-12.42
	8	5.230	4.799	8.24
	16	2.579	2.147	16.75

5.5 Conclusion

In this chapter, an efficient synchronous parallel progressive radiosity algorithm based on Hemicube Merging corresponding to Level 2 of the classification is presented. Efficient communication schemes are proposed for increasing the performance of the parallel solution.

The performance of the Hemicube Merging Scheme increases with increasing number of patches with a given hemicube resolution. Furthermore, as this scheme follows the shooting patch sequence of the sequential algorithm, the parallel algorithm performs well for high tolerance values without increasing the number of shooting patches. Hence, this scheme can be used in order to obtain approximated solutions for complex scenes, which requires less number of shooting patches than the Patch Circulation Scheme.

Chapter 6

Conclusion

In this thesis, the progressive refinement radiosity method has been investigated for parallelization on multicomputers. Two schemes based on two levels of parallelism have been proposed and implemented on hypercube-connected and ring-connected multicomputers. The proposed parallel schemes utilize a synchronous type of parallelism based on static task assignment. This chapter includes the concluding remarks and the comparison of the proposed schemes.

The radiosity method correctly approximates the light distribution among the objects in an environment, because it is based on physical principles. Chapter 2 introduced the radiosity method, and summarized the further improvements of the method. The radiosity method requires excessive amount of computation. The method slows down significantly when highly specular surfaces are included in the environment. Also, non-polygonal objects are not easily handled by the method.

Chapter 3 introduced the parallelism preliminaries and design criteria for parallel computer graphics applications. Although the given list of criteria is not complete, it includes the general concepts of parallel computer graphics algorithm design. Then, the previous work on parallelization of the progressive refinement radiosity method was classified and presented.

Chapter 4 presented the first proposed parallelization scheme based on patch data circulation. The scheme is a modification to the sequential progressive radiosity method. In this method, P patches are selected as the shooting patches and processed simultaneously. The synchronous parallelism has shown to be scalable for even very simple topologies such as the ring, whereas the performances of the asynchronous parallel solutions that were proposed in the literature downgrade significantly on simple topologies with a small number of links per processor. The synchronous parallelism additionally allows to maintain shooting patch selection coherence, because the light energy distributed by the shooting patches at an iteration reflects the effect of the distributions at the previous iterations. However, load imbalance problem is a critical issue for synchronous parallelism. Experimental results have shown that scattered decomposition of the scene geometry yields adequate load balance during parallel hemicube production computations. The new contribution computation scheme proved to decrease the total volume of communications by an asymptotical factor. Modification of the original progressive radiosity for the sake of efficient parallelization is experimentally found to yield good results for relatively small number of processors. The performance of this modification is expected to increase with decreasing tolerance values which necessitate larger number of iterations for convergence.

In Chapter 5, the second scheme for the parallelization of the progressive radiosity method, the Hemicube Merging Scheme, has been presented. The scheme is based on the environment-projection Hemicube Method for form-factor computation, which is the most efficient method that can be performed with the existing hardware. The scheme processes one shooting patch at a time and is based on a divide-and-conquer approach. First, each processor computes the projection of a different local subset of the patch data, then the resulting hemicubes are merged in order to compute the final hemicube. Load balance qualities of the decomposition schemes are compared, and scattered decomposition achieved adequate load balance. The total volume of communication and the extra computation in order to merge the computed hemicubes is a critical problem for this scheme. A new communication scheme that decreases the total amount of extra computations and the volume of communications by an asymptotical factor, has been presented. The Hemicube Merging Scheme is shown to be scalable with even simple topologies such as the ring.

For both schemes, we have tried to minimize the memory requirements, by avoiding the duplication of the scene in each processor's local memory. Thus, complex scenes which could not be handled by a single processor can be processed. The results show that we can reach a good speed-up for any hemicube size, and any number of patches, by using one of the proposed schemes. The Hemicube Merging Scheme performs more efficiently for small hemicube size and large scenes, whereas the Patch Circulation Scheme works well for large hemicube sizes and small scenes, comparatively. The modification in the Patch Circulation Scheme increases the total number of shooting patch selections in comparison with the original sequential algorithm, and hence the Hemicube Merging Scheme. This increase has shown to become more crucial with greater tolerance values. Thus, the Hemicube Merging Scheme works better for large tolerance values, whereas the Patch Circulation Scheme can reach good performance with smaller tolerance values.

For future work, better patch decomposition schemes based on a predetermined heuristic that provide better load balance can be investigated. The total volume of communications can be decreased if the patch-to-patch visibility is pre-computed and only the required patches are communicated and processed. The parallel solutions combined with adaptive subdivision of the input geometry should be investigated. Other accurate form-factor computation techniques (such as analytical form-factor, Wallace's ray tracing) present new problems such as load balancing. These techniques should be investigated for parallelism. Other methods such as hierarchical radiosity, discontinuity meshing, should also be considered for parallelization for future work.

Bibliography

[1] Aykanat, Cevdet, Tolga K. Çapın, Bülent Özgüç, "Progressive Refinement Radiosity Based on Patch Data Circulation for Multicomputers", unpublished manuscript, 1993.

[2] Baum, Daniel R., John R. Wallace, Donald P. Greenberg, "The Back Buffer: An Extension of the Radiosity Method to Dynamic Environments", The Visual Computer, Vol.2, No.5, pp 298-306, 1986.

[3] Baum, Daniel R., Holly E. Rushmeier, James M. Winget, "Improving Radiosity Solutions Through the Use of Analytically Determined Form-Factors", Proceedings of SIGGRAPH '89. In Computer Graphics Vol.23, No.3, July 1989, pp325-334.

[4] Baum, Daniel R., James M. Winget, "Real Time Radiosity Through Parallel Processing and Hardware Accelaration", Proceedings of the 1990 Symposium on Interactive 3D Computer Graphics, In Computer Graphics, Vol.24, No.2, 1990, pp 67-75.

[5] Baum, Daniel R., Stephen Mann, Kevin P. Smith, James M. Winget, "Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation og Accurate Radiosity Solutions", Proceedings of SIGGRAPH '91. In Computer Graphics, Vol.25, No.4, July 1991, pp 51-60.

[6] Bouatouch, Kadi, Daniel Menard, Thierry Priol, "Parallel Radiosity Using a Shared Virtual Memory", Proceedings of ATARV'93, Ankara, Turkey, 1993, pp 71-83.

[7] Bu, J. and E.F. Deprette, "A VLSI System Architecture for High Speed Radiative Transfer in Three-Dimensional Image Synthesis", The Visual Computer, Vol.5, pp 131-133, 1989.

[8] Buckalew, C., Donald Fussell, "Illumination Networks: Fast Realistic Rendering with General Reflectance Functions", Proceedings of SIGGRAPH '89. In Computer Graphics, Vol.23, No.3, July 1989, pp 89-98.

[9] Bui-Tuong, Phong, "Illumination of Computer Generated Pictures", Communications of the ACM, Vol.18, No.6, June 1975, pp 311-317.

[10] Campbell, A.T. III, Donald S. Fussell, "Adaptive Mesh Generation for Global Diffuse Illumination", Proceedings of SIGGRAPH '90. n Computer Graphics Vol.24, No.4, August 1990, pp155-164.

[11] Çapın, Tolga K, Cevdet Aykanat, Bülent Özgüç, "Progressive Refinement Radiosity on Ring-Connected Multicomputers", Proceedings of Visualization'93 Parallel Rendering Symposium (San Jose, California), 1993.

[12] Chalmers, Alan G., Derek J. Paddon, "Parallel Processing of Progressive Refinement Radiosity Methods", Proceedings of the Second Eurographics Workshop on Rendering, Barcelona, Spain, May 1991.

[13] Chen, Shenchang Eric, "Incremental Radiosity: An Extension of Progressive Radiosity to an Interactive Image Synthesis System", Proceedings of SIGGRAPH '90. In Computer Graphics, Vol.24, No.4, August 1990, pp 135-144.

[14] Cohen, Micheal F., Donald P. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments", Proceedings of SIGGRAPH '85 (San Fransisco, California, July 1985). In Computer Graphics, Vol.19, No.3, 1985, pp 31-40.

[15] Cohen, Micheal F., Donald P. Greenberg, David S. Immel, Philip J. Brock. "An Efficient Radiosity Approach for Realistic Image Synthesis, IEEE Computer Graphics and Applications, Vol. No., March 1986, pp 26-35.

[16] Cohen, Michael F., Shenchang Eric Chen, John R. Wallace, Donald P. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Generation", Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1988). In Computer Graphics, Vol.22, No.4, 1988, pp 75-84.

[17] Dekel, E., D. Nassimi, S. Sahni, "SIAM Journal on Computing", Vol.10, No.4, 1981, pp 657-675.

[18] Drucker, Steven M. and Peter Schröder, "Fast Radiosity Using a Data Parallel Architecture", Proceedings of the Third Eurographics Workshop on Rendering, Bristol, England, May 1992, pp 247-258.

[19] Feda, Martin, Werner Purgathofer, "Progressive Refinement Radiosity on a Transputer Network", Proceedings of the Second Eurographics Workshop on Rendering, Barcelona, Spain, May 1991.

[20] Flynn, MJ, "Very High-Speed Computing Systems", Proceedings of IEEE, Vol.54, Dec. 1966, pp 1901-1909.

[21] Foley, James D., Andries van Dam, Steven K. Feiner, John F. Hughes, "Computer Graphics: Principles and Practice (Second Edition)", Addison-Wesley Publishing Company.

[22] Goral, Cindy M., Kenneth E. Torrance, Donald P. Greenberg, Bennett Battaile, "Modelling the Interaction of Light Between Diffuse Surfaces", Proceedings of SIGGRAPH '84 (Boston, Massachusetts). In Computer Graphics, Vol.18, No.3, July 1984, pp 213-222.

[23] Gouraud, H., "Continuous Shading of Curved Surfaces", IEEE Transactions on Computers, C-20(6), June 1971, pp 623-629.

[24] Guitton, P., J. Roman, C. Schlick, "Two Parallel Approaches for a Progressive Radiosity", Proceedings of Second Eurographics Workshop on Rendering, Barcelona, Spain, May 1991.

[25] Hanrahan, Pat, David Salzman, Larry Auperle, "A Rapid Hierarchical Radiosity Algorithm", Proceedings of SIGGRAPH '91. In Computer Graphics, Vol.25, No.4, July 1991, pp 197-206.

[26] Heckbert, Paul S., "Adaptive Radiosity Textures for Bidirectional Ray Tracing", Proceedings of SIGGRAPH '90. In Computer Graphics, Vol.24, No.4, August 1990, pp 145-154.

[27] Immel, David S., Michael F. Cohen, "A Radiosity Method for Non-Diffuse Environments", Proceedings of SIGGRAPH '86 (Dallas, Texas). In Computer Graphics, Vol.20, No.4, 1986, pp 133-142.

[28] Jessel, J.P., M. Paulin, R. Caubet, "An Extended Radiosity Using Parallel Ray-traced Specular Transfers", 1991.

[29] Lischinski, Dani, Filippo Tampieri, Donald P. Greenberg, "Discontinuity Meshing for Accurate Radiosity", IEEE Computer Graphics and Applications, Vol. No., November 1992, pp 25-39.

[30] Nishita T., E. Nakamae, "Continuous Tone Representation of Three-Dimensional Objects Taking Account of Shadows and Interreflection", Proceedings of SIGGRAPH '85. In Computer Graphics Vol.19, No.3, pp 23-30, 1985.

[31] Price, Martin and Greg Truman, "Radiosity in Parallel", Proceedings of the First International Conference on Applications of Transputers, 1989.

[32] Puech, Claude, Francois Sillion, Cristophe Vedel, "Improving Interaction with Radiosity-based Lighting Simulation Programs", Proceedings of the 1990 Symposium on Interactive 3D Computer Graphics, In Computer Graphics, Vol.24, No.2, 1990, pp 51-57.

[33] Purgathofer Werner, M. Zeiller, "Fast Radiosity by Parallelization", Proceedings of Eurographics Workshop on Photosimulation, Realism and Physics, 1990, pp 173-185.

[34] Recker, Rodney J, David W. George, Donald P. Greenberg, "Acceleration Techniques for Progressive Refinement Radiosity", Proceedings of the 1990 Symposium on Interactive 3D Computer Graphics, In Computer Graphics, Vol.24, No.2, 1990, pp 59-66.

[35] Siegel, R. and J.R. Howeol, "Thermal Radiation Heat Transfer", Hemisphere Publishing Corp., Washington DC, 1981

[36] Shao, Min-Zhi, Qun-Sheng Peng, You-Dong Liang, "A New Radiosity Approach by Procedural Refinements for Realistic Image Synthesis", Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1988). In Computer Graphics, Vol.22, No.4, 1988, pp 93-101.

[37] Smits, Brian E., James R. Arvo, David H. Salesin, "An Inportance-Driven Radiosity Algorithm", Proceedings of SIGGRAPH '92, In Computer Graphics, Vol.26, No.2, July 1992, pp 273-282.

[38] Sutherland, I.E., G.W. Hodgman, "Reentrant Polygon Clipping", Communications of the ACM, Vol.17, No.1, January 1974, pp 32-42.

[39] Sutherland, I.E., R.F. Sproull, R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", ACM Computing Surveys, Vol.6, No.1, March 1974, pp 1-55.

[40] Teller, Seth J., Carlo H. Sequin, "Visibility Preprocessing for Interactive Walkthroughs", Proceedings of SIGGRAPH '91, In Computer Graphics, Vol.25, No.4, July 1991, pp 61-69.

[41] Varshney, Amitabh and Jan F. Prins, "An Environment Projection Approach to Radiosity for Mesh-Connected Computers", Proceedings of the Third Eurographics Workshop on Rendering, Bristol, England, May 1992, pp 271-281.

[42] Wallace, John R., Micheal F. Cohen, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods", Proceedings of SIGGRAPH '87, In Computer Graphics, Vol.21, No.4, 1987, pp 311-320.

[43] Wallace, John R., Kells A. Elmquist, Eric A. Haines, "A Ray Tracing Algorithm for Progressive Radiosity", Proceedings of SIGGRAPH '89, In Computer Graphics Vol.23, No.3, July 1989, pp 315-324.

[44] Whitted, T., "An Improved Illumination Model for Shaded Display", Communications of the ACM, Vol.26, No.6, 1980, pp 342-349.

Appendix A

Scene Images

This appendix contains the final images of the scenes for testing the performances of the parallel algorithms.

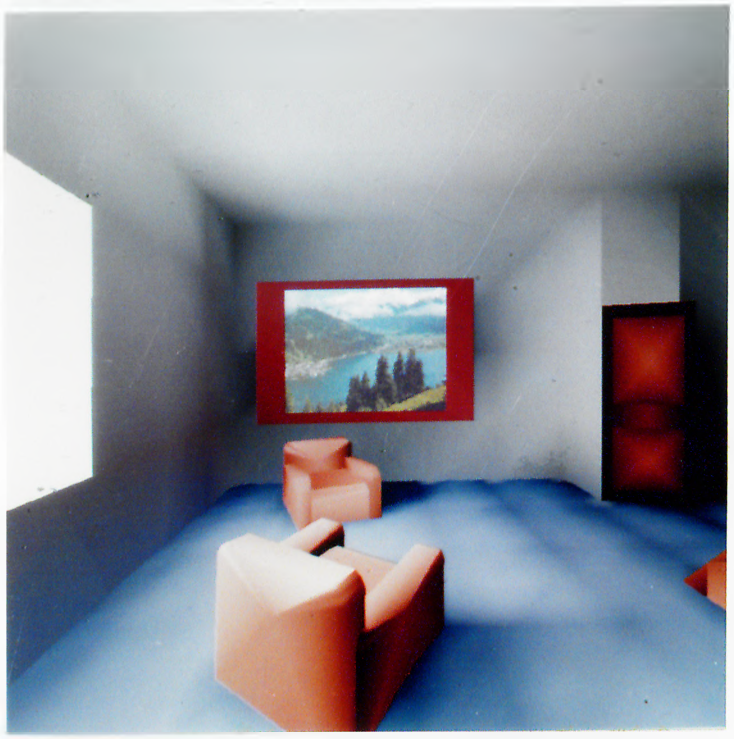


Figure A.1. House Scene Data with 5648 Patches

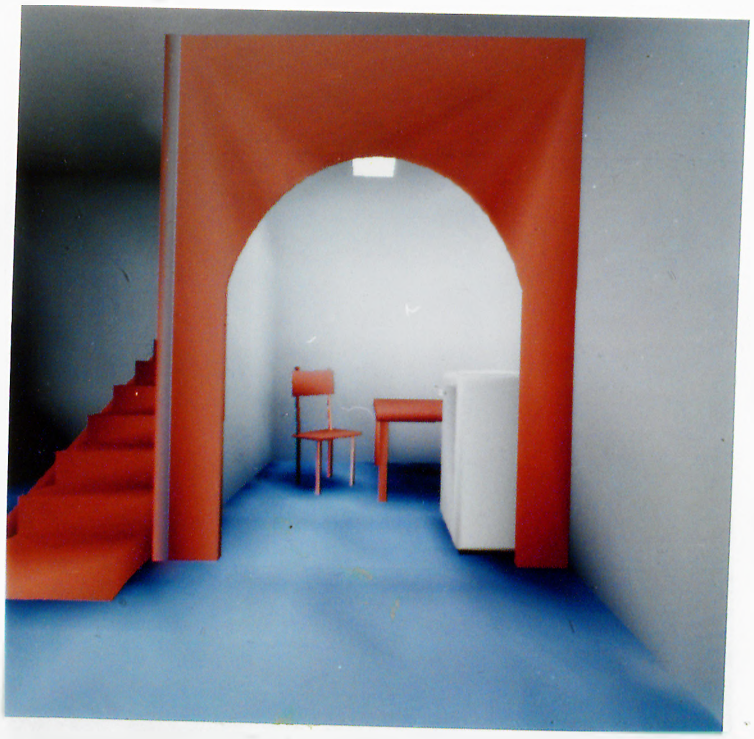


Figure A.2. Another view of the house scene data

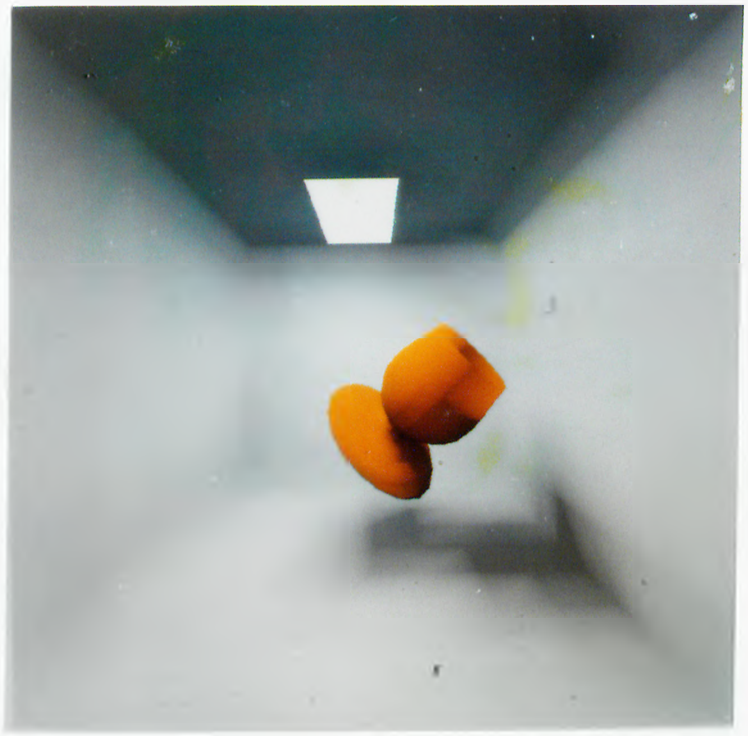


Figure A.3. A Frame from an animation sequence (3424 patches)

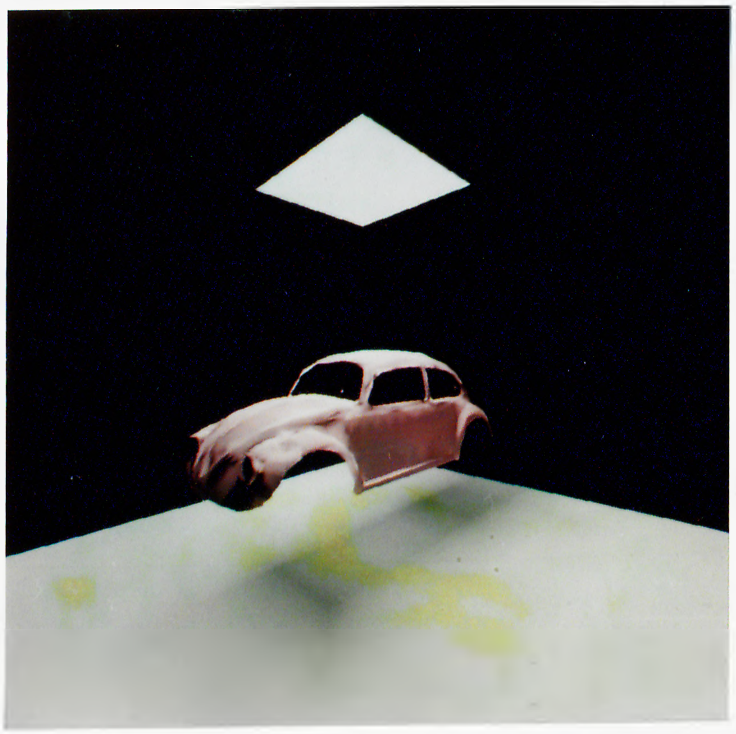


Figure A.4. Image of a Volkswagen Data