# PARALLELIZATION OF HIERARCHICAL RADIOSITY ALGORITHMS ON DISTRIBUTED MEMORY COMPUTERS

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
Ahmet Reşat Şireli
January, 1999

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.

Asst. Prof. Attila Gürsoy (Supervisor)

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

Assoc. Prof. Cevdet Aykanat

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.

Asst. Prof. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of Institute of Engineering and Science

# ABSTRACT

PARALLELIZATION OF
HIERARCHICAL RADIOSITY ALGORITHMS
ON DISTRIBUTED MEMORY COMPUTERS

Ahmet Reşat Şireli
M.S. in Computer Engineering and Information Science
Supervisor: Asst. Prof. Attila Gürsoy
January 1999

Computing distribution of light in a given environment is an important problem in computer-aided photo-realistic image generation. Radiosity method has been proposed to address this problem which requires an enormous amount of calculation and memory. Hierarchical radiosity method is a recent approach that reduces these computational requirements by careful error analysis. It has its idea from the solution methods of N-body problems. Although hierarchical approach has greatly reduced the amount of calculations, satisfactory results still cannot be obtained in terms of processing time. Exploiting parallelism is a practical way to reduce the computation time further. In this thesis, we have designed and implemented a parallel hierarchical radiosity algorithm for distributed memory computers. Due to its highly irregular computational structure, hierarchical radiosity algorithms do not yield easily to parallelization on distributed memory machines. Dynamically changing computational patterns of the algorithm cause severe load imbalances. Therefore, we have developed a dynamic load balancing technique for the parallel hierarchical radiosity calculation.

*Keywords*: Realistic Image Generation, Parallel Hierarchical Radiosity, Dynamic Load Balancing.

# ÖZET

DAĞITIK BELLEKLİ BİLGİSAYARLARDA
SIRADÜZENSEL IŞIMA ALGORİTMALARININ
PARALELLEŞTİRİLMESİ

Ahmet Reşat Şireli
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yard. Doç. Dr. Attila Gürsoy
Ocak 1999

Işığın verilen ortam içerisinde dağılımını hesaplamak bilgisayar destekli gerçeğe uygun görüntü üretiminde önemli bir problemdir. Işıma metodu, bu aşırı bir miktarda hesap ve hafıza gerektiren problemin çözümü için önerilmiştir. Sıradüzensel ışıma metodu, bu işlemsel gereksinimleri dikkatli hata analizi sonucu azaltan nihai yaklaşımlardan biridir. Fikrini N-gövde probleminin çözüm metodlarından almıştır. Sıradüzensel yaklaşım işlemlerin miktarını büyük ölçüde azaltmış olmasına rağmen, zaman bakımından tatminkar sonuçlar hala elde edilememektedir. Paralellikten faydalanmak işlemsel sürenin daha da azaltılması için pratik bir metoddur. Bu tezde, dağıtık bellekli bilgisayarlar için bir paralel sıradüzensel ışıma algoritması tasarladık ve uyguladık. Aşırı düzensiz işlemsel yapısı yüzünden sıradüzensel ışıma algoritmaları dağıtık bellekli bilgisayarlar üzerinde paralelleştirilmesi kolay olmamaktadır. Algoritmanın dinamik olarak değişen işlemsel örüntüleri birçok yük dengesizliklerine sebep olmaktadır. Bu yüzden paralel sıradüzensel ışıma algoritmamız için bir dinamik yük dengeleme tekniği de geliştirdik.

*Anahtar sözcükler*: Gerçeğe Uygun Görüntü Üretimi, Paralel Sıradüzensel Işıma, Dinamik Yük Dengeleme.

# ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest thanks and gratitude to my advisor Asst. Prof. Attila Gürsoy for his patient supervision of this thesis.

I am grateful to Assoc. Prof. Cevdet Aykanat and Asst. Prof. Uğur Güdükbay for reading the thesis and for their instructive comments. I would like to acknowledge the financial support of TÜBİTAK under the grant EEEAG-247.

Special thanks go also to Asst. Prof. Uğur Güdükbay for providing the substance of this research work and to Assoc. Prof. Cevdet Aykanat for permitting us to use the machine Parsytec CC (through ITDC 204-82166 and TÜBİTAK EEEAG-160).

I would also like to thank my family for their encouragement; my sister Filiz, my friends Önder, Seher and Yücel for their moral support; and finally all other friends who contributed this study.

**To my family, to infinity and beyond ...**

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Photo-realistic image generation is a difficult and time-consuming problem of computer graphics. Difficulty arises because of the necessity of simulating the real-world lighting events with considering all the possible physical effects. Formulation may not be enough to express events completely. Time-consuming property is due to its enormous amount of calculation requirement which is reversely proportional with the quality of image. While wishing to develop a real-time interactive image generator, we still have to wait for minutes even for simple scenes. Despite of all these negative factors, it still preserves its attractiveness for the researchers. Combined with animation, image generation is a very promising subject that has too many application areas such as simulating, training, design and manufacturing, telecommunications, medicine, information visualization. Although it may seem as a very difficult problem today, technological improvements will never stop and one day we will be able to use it in our daily life.

An important problem in achieving realistic image generation is computing the distribution of light in an environment. Radiosity approach has been proposed to solve this problem [GTGB84] whose principles are based on a research area of thermodynamics, heat transfer. Instead of heat transfer, we consider energy transfer between surfaces, in radiosity method. The method requires all of the surfaces in scene to exchange light energy according to some configuration factors. Calculation of these configuration factors is the most important and time-consuming part of radiosity method.

Hierarchical radiosity is a recent approach that reduces computational requirements by careful error analysis. It has its idea from the solution methods of N-body problems. N-body solution methods approximate the interactions between well separated groups of objects by a single interaction. Consequently many expensive calculations that have little effect on the accuracy of the solution can be avoided. Hierarchical radiosity applies this approach by eliminating the interactions of surfaces that do not effect the accuracy of the overall solution. Criterion of efficiency of an interaction is the radiant flux carried by it. As stated in [CW93], hierarchical techniques reduce the complexity of radiosity computations from $O(n^2)$ to $O(n+k^2)$, where $k$ is the number of input surfaces, and $n$ is the total number of resultant elements in an environment ($n \gg k$).

Although hierarchical approach has greatly reduced the amount of calculations, satisfactory results still cannot be obtained in terms of time. As we mentioned in the beginning, none of the methods can achieve image production in a reasonable time with a reasonable accuracy yet. Parallel processing is one of the most practical ways to reduce the computation time further. Due to its highly irregular computational structure, hierarchical radiosity algorithms do not yield easily to parallelization on distributed memory machines. In this thesis, we investigate the feasibility of parallel processing for hierarchical radiosity. This work consists of three main parts:

- design of a parallel algorithm and dynamic load balancing mechanisms for radiosity,
- design and implementation of a parallel object-oriented program, and
- performance study and impact of various design decisions.

The parallel algorithm is based on the sequential hierarchical radiosity algorithm presented in [HSA91]. The algorithm we proposed can also be extended to the recent improvements on this field, such as clustering, lazy linking etc. The object-oriented implementation has been done using Charm++ programming environment. Charm++ is a parallel object-oriented language developed at UIUC [KK93]. As part of this thesis work, we also ported Charm++ to the Parsytec CC[1] distributed memory machine to conduct the performance studies.

The thesis is organized as follows. Section 2 presents a study on asynchronous message handling strategies. First, we make an overview of asynchronous message handling strategies, and then present a study about porting Charm++/Converse to Parsytec multicomputer [ACSG98]. The objective of this study is to investigate performance of the system that we are going to use for the rest of the study.

Section 3 gives a description of the radiosity problem and the hierarchical model. Concepts of the problem and solution methods are discussed briefly. We also present a design of sequential version of the hierarchical radiosity algorithm including some implementation issues. This study is performed in order to separate the issues related with parallelization as much as possible.

Section 4 discusses parallelization of the hierarchical radiosity algorithm in detail. After an overview of related work, we explain and discuss our design and implementation issues. We also discuss performance issues such as load balancing, at the end of the section.

In Section 5, this work is evaluated with performance results. Timings are presented for sample scenes and impacts of some design decisions on performance are analyzed.

The thesis finishes by concluding the studies in the last section. Including the critique of our implementation, future work is also provided. The goals met are stated, those unmet are discussed.

[1]Parsytec CC is a registered trademark of Parsytec, Inc.

# Chapter 2

# Asynchronous Message Handling

In this chapter, we make an overview of asynchronous message handling strategies and present a study about porting Charm++, which is a parallel object-oriented language, to Parsytec multicomputer. Porting Charm++ to a new machine requires reimplementation of some parts of its machine dependent layer. This machine layer in Charm++ belongs to a component called Converse (an interoperable framework for parallel programming). Therefore, we have implemented machine dependent layer of Converse for the Parsytec computer.

The motivation behind this work is to use advantages of Charm++ programming environment not only for the parallel implementation of hierarchical radiosity, but also for other applications that might be developed later. The details of this port and the advantages of Charm++ environment is discussed in the following sections.

## 2.1 Asynchronous Message Handling

The major source of performance degradation in message passing parallel computers is the delays due to communication which is an inevitable requirement.

These delays are not only due to latencies in the communication network. A significant portion of it is due to software overhead to handle sending and receiving messages from user level to the hardware level. Due to its unavoidable and unnegligible impact on performance, parallel system designers spend great efforts to build systems which minimize message processing overhead, in addition to hardware improvements. Asynchronous message handling is such a study which aims to reduce this delay by handling messages efficiently at the receiving side. This is achieved by overlapping communication with computation and providing an efficient mechanism to handle the incoming message.

Asynchronous message handling is useful in irregular applications in which sending and receiving of messages are performed in a non-deterministic order. This is frequently the case in applications of parallel object-oriented systems. Processes in such systems do not operate in a synchronized manner and therefore cannot know when to expect incoming messages. It is possible to receive messages synchronously by regularly polling the network, however achieving good performance with a polling-based approach is not easy [LBB97]. Another way is using interrupts to deliver the incoming message. However, as discussed in [LBB97], polling network is much cheaper than using interrupts.

In contrast to message passing, in asynchronous message handling model message reception results in invocation of a function, which is called message handler. Expressiveness of the message handler is an important factor in achieving good performance. There are several asynchronous message handling systems developed (e.g. [ECGS92], [PLC95]) and they vary in the way the handlers are executed, the expressiveness of the model etc. Less expressive models restrict functions of a message handler in order to be more efficient, whereas more expressive models do not put restrictions on message handlers in order to be more expressive. The less the expressiveness, the more the efficiency is.

Active messages model [ECGS92] is an example of restrictive model. It restricts blocking of message handlers. Also, message handlers cannot allocate memory or initiate communications to other processors. Because message handlers do not have their own execution context. Therefore, there is no a different stack, no thread definition and no thread switching. Although these features make the model very effective, they significantly complicate programming. Other mechanisms include single-threaded upcall and pop-up threads, which disallow all blocking by letting message handlers use locks to synchronize their shared-data accesses.

Asynchronous message handling deals with only reducing the software overhead of sending and receiving a message. However, the physical reality dictates that accessing a remote information will always be slower than accessing the local one. Hence, in addition to reducing the delays, overlapping these delays due to remote data access with useful computation is also important for improving performance of parallel systems. Message driven execution is a promising model in this sense [KG95]. In message driven execution, there are many objects per processor. When a message arrives for an object, the object is eventually activated with the message. Although message driven execution sounds similar to asynchronous message handling, it requires a scheduler. In asynchronous message handling the invocation of handlers is not under the control of user and they lack user level scheduler. Converse [KBJK96] is one of the early implementations of a parallel programming system that combines scheduler with asynchronous message handling. A brief information about Converse is given in the next section.

In the rest of the chapter, we will discuss porting Converse to Parsytec CC parallel computer.

## 2.2 Converse and Its Machine Interface

Generally speaking, Converse is a library of subroutines for parallel processing. In contrast to traditional receive based message passing, Converse [KBJK96] is a message-driven parallel programming language which combines user level scheduler with asynchronous message handling.

Converse is a portable language which has been implemented on various machines such as Origin 2000, IBM SP3, CM5, Cray T3E etc. Converse Machine Interface (CMI) contains functions which must be implemented to port

Converse to a parallel computer. The CMI module is responsible for process creation and coordination at the communication and some other utilities such as timers required for portability. The functions contained in CMI can be grouped under the following headers:

- **Message sending functions:** `CmiSyncSend()`, `CmiAsyncSend()` and their variants.
- **Message broadcasting functions:** `CmiSyncBroadcast()`, `CmiAsyncBroadcast()` and their variants.
- **Initialization/termination functions:** `ConverseInit()`, `ConverseExit()`
- **Neighbor determination functions:** `CmiMyNode()`, `CmiNumNodes()`, `CmiMyRank()`, `CmiRankOf()` and a few more.
- **Memory allocation/free functions:** `CmiAlloc()`, `CmiFree()`, `CmiSize()`
- **Handler related functions:** `CmiSetHandler()`, `CmiGetHandler()`, `CmiGetHandlerFunction()`

In this work, we consider minimal interface of the message sending functions. Complete information about the CMI functions is available in [CON96].

## 2.3 The Underlying System, Parsytec CC

The Parsytec CC system is a parallel computer manufactured by Parsytec GbmH in Aachen, Germany. It is based on distributed memory MIMD architecture. All nodes of Parsytec CC system run the AIX operating system with EPX, Embedded Parix on top. EPX provides set of functions to build and to use a communication network and to define suitable routines managing data operations.

There are three types of communication available in EPX. These are namely synchronous virtual link bound communication, synchronous random communication and asynchronous link bound communication. PVM is also available in the system.

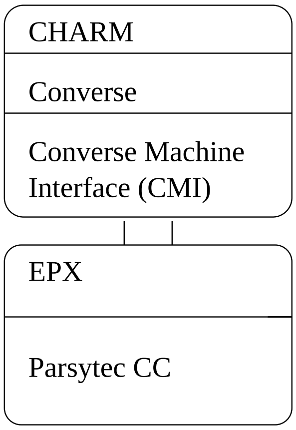


Figure 2.1: Port environment.

## 2.4 Implementation

Porting Converse to Parsytec was a study of bridging the gap between CMI and EPX (see Figure 2.1). We implemented the CMI machine interface on top of EPX, which is the native message passing library of Parsytec CC machine. However, the functionality of EPX message passing primitives is not sufficient to express all the CMI primitives directly. One of the problems is that synchronous message passing functions of EPX blocks the caller until the tail of message enters the network. In Converse, however, the control must return to the caller just after the send function call. Asynchronous primitives of EPX are not compatible with CMI asynchronous functions either. The problem is that in EPX a sender processor cannot detect whether a particular message is reached the destination or not, during asynchronous communication. Rather, it can only check if there exists any message on the link that the message is sent through. Another problem is about the size of messages that are to be transmitted. EPX receive primitives require size of messages in advance. But EPX does not support functions to evaluate the size of the incoming message, rather it is maintained by the programmer. On the other hand Converse does not know which message is going to be received next. Therefore we developed an efficient mechanism to receive arbitrary size messages.

To overcome problems related with asynchronous message sending and receiving, we have designed a layer containing threads and message queues. That is on each processor, the main thread (the process) executes the scheduler of

Converse. The messages are sent and received by separate threads which are responsible for communication, within the process. These threads perform message passing by calling EPX functions. So there are sender and receiver threads. A receiver thread checks the incoming transmission links with which it's associated, and as it detects a message on any of those links, it receives the message using `RecvLink()` primitive and appends it into the receive queue of its owner processor. When a message is detected in receive queue, it is picked up rapidly by the scheduler using `CmiGetNonLocal()` function and associated handler function is invoked with necessary parameter that is with the message itself. Sender threads are responsible for sending messages which are picked up from send queues, to related processors. Respectively when a message is detected in any send queue, it is transmitted into the network using EPX `SendLink()` primitive.

We had some alternatives for the number of sender and receiver threads. These were:

- 1-sender 1-receiver
- $N$-sender 1-receiver
- 1-sender $N$-receiver
- $N$-sender $N$-receiver

where $N$ is *number of nodes - 1* . We found it better to use $N$ send queues and $N$ sender threads, which each thread associated with a separate message queue and a processor, in order to avoid contention on queues on concurrent send requests to different target processors. For the receiving threads, however, we needed to maintain a single queue for incoming messages to keep the First-In-First-Out (FIFO) order. Using one receive thread results better than using $N$ number of receive threads, since in the latter case the threads will waste CPU time sharing the single receive queue. The one-and-only receive thread uses the `SelectList()` function of EPX in order to wait messages from all transmission links. In order to make comparison, we also implemented the $N$ receiver version. The versions are discussed in the next sections.

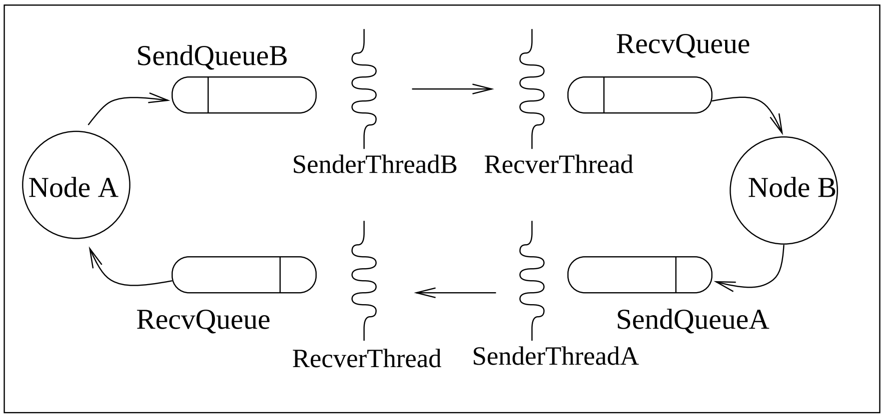


Figure 2.2: Communication model of $N$-sender 1-receiver version.

For message size problem, we developed an efficient mechanism to receive arbitrary size messages by defining a standard message size SIZE. The mechanism works as follows: The messages that are less than SIZE bytes are sent directly without an extra work. Longer messages are sliced into 2 parts. Size information is appended to the last 10 bytes of the first part and totally SIZE+1 bytes are sent in the first communication. The rest of the message is sent in the second communication as a whole. When the receiver receives a message with size of 1 to SIZE bytes, it understands that the whole message has been received. If the message is SIZE+1 bytes, it reads the size information from the last of 10 bytes of the message, allocates that much buffer, copies the first part of the message to this buffer and receives the rest of the message to be placed next to the first part in the buffer.

### 2.4.1 $N$-sender 1-receiver Version

The first step to implement an EPX program is to select an appropriate topology. Converse is a system which requires direct communication between any two nodes. EPX's virtual topology library supports clique topology which satisfies this requirement. However, we observed that creating completely connected topology explicitly with `CreateLink()` function, gives better performance in terms of communication speed. And also the function `SelectList()` we used requires links to be created explicitly.

The communication model is seen on Figure 2.2. The model is valid for every pair of nodes. Each node has $N$ sender threads and $N$ send queues each associated with a particular processor. The receive thread and the receive queue is only one in each node.

In order to prevent the sender threads to be active and using CPU while there is no message to be sent, we used semaphore mechanism. The idle threads wait in a lock to acquire a semaphore which will be released when there is a message to be sent. Also, exclusive usage of message queues are provided by semaphores.

The function we used to decrease the number of receiver threads is the `SelectList()` call, which provides receiving message from any node. It returns the identifier of the node which tries to send a message to that node. A receiver thread waits for new messages to be received from any processor by calling this function. As soon as the arrival of a message, the thread appends the message to the tail of message queue.

A sender thread waits for the messages to be sent in a blocked position. It tries to acquire a semaphore which will be released when a new message is appended to its message queue. When it acquires the semaphore, it takes the appended message from the queue and send it. The sender threads are used only for asynchronous send requests. To speed up the communication, synchronous send requests are performed by directly calling `SendLink()` function of EPX. These two different approaches may lead a change in the order of messages. In order to preserve the order, the synchronous send requests are performed only after providing that there are no messages to be sent waiting in the queues.

Let's see how a send request is performed, on the nodes presented in the Figure 2.2. Node $A$ wants to send a message to node $B$ using asynchronous communication. It appends the message to the message queue, *sendQueueB* and releases *semaphoreB* to activate *senderThreadB* which waits as having been blocked. The call returns immediately with the pointer of queue element where the message is kept. By the help of this pointer, it can be checked whether the communication is done or not. *SenderThreadB* picks up the message from its queue immediately and invokes a send call to node $B$. *ReceiverThread* receives the message rapidly and appends it to the *recvQueue*. As soon as the communication is completed, the message is removed from the message queue. At the end, the scheduler of node $B$ calls `CmiGetNonLocal()`, picks up a message from the received message queue *recvQueue* and processes it.

Broadcast operation is also realized by threads. The message is appended to all of the message queues of the sender threads one by one. For synchronous broadcast operation, we wait for the end of all sending operations. However, for asynchronous type, we return the control to the caller procedure immediately. Sender threads pick up the message rapidly and try to send them to all other nodes of the system. In asynchronous broadcast operation, we have a chance to overlap computation and communication, since there is no a blocking type operation.

### 2.4.2 $N$-sender $N$-receiver Version

Different from the 1-receiver version, there are $N$ receiver threads in each node. This version is implemented to compare the overhead of using more receiver threads with using `SelectList()` call. In this version no `SelectList()` is called since there are already $N$ receiver threads instead of that call. Since all the receiver threads uses only one message queue, accesses to this queue is restricted. Mutual exclusion is provided by semaphore mechanism.

## 2.5 Performance Evaluation

In order to measure and compare performances of the asynchronous message handling (Converse) and the message passing libraries which are available on Parsytec CC system, namely EPX [EPX95] and PVM [GBD+94], we conducted a performance study. The study included a simple ring communication algorithm for measuring message latency as observed by the programmer, and the k-to-all broadcast algorithm. Message-driven execution has advantages if multiple messages arrive in an unpredictable order and if they can be processed not in a strict order. K-to-all broadcast where multiple messages arrive in an unpredictable order demonstrates this advantage. The algorithms have been implemented using EPX, PVM, and Converse systems as efficient as possible for each case.

### 2.5.1 Communication via Ring

In the ring program, the processors are connected such that they form a ring topology, and they pass messages from their predecessors to their successors. When the processor that has sent the message first, receives the message from its predecessor, the ring computation finishes. The communication is regular in the sense that there is only one message and each processor knows from where to receive and to where to send the message. Table 2.1 shows the round-trip time for messages of different length for Converse, PVM and EPX versions of ring. The EPX version is slightly faster than PVM and Converse ones because Converse and PVM runtime systems are built on EPX and the difference is the software overhead introduced by Converse and PVM systems. However, as the messages get larger, Converse results start becoming better than PVM and get closer to EPX results. So, Converse incurs negligible overhead for having the capability of asynchronous message handling. The next example, k-to-all broadcast shows a significant performance improvement in case of handling multiple messages.

### 2.5.2 K-to-all Broadcast Communication

In k-to-all broadcast, k processors simultaneously perform a one-to-all broadcast of m-word messages. The broadcast operation is implemented by forming a spanning tree covering all the processors where the root node is the initiator of the broadcast. Table 2.2 shows the completion time of k-to-all broadcast. In this case, there are multiple broadcast operations going on concurrently. Therefore, in the EPX and PVM version, these messages are handled in a fixed order however in Converse version the messages trigger the appropriate operations as they arrive. As shown in the figure, the Converse version is significantly

Table 2.1: Timings for ring program (in msec).

| message size | processor # | $N$-sender $N$-receiver | $N$-sender 1-receiver | PVM | EPX |
|---|---|---|---|---|---|
| 64 bytes | 2 | 1.73 | 1.73 | 1.68 | 0.81 |
| | 4 | 3.55 | 3.56 | 3.39 | 1.18 |
| | 8 | 7.21 | 7.25 | 6.61 | 2.41 |
| | 16 | 14.66 | 15.08 | 13.39 | 4.88 |
| 64 Kbytes | 2 | 9.84 | 9.84 | 11.13 | 7.13 |
| | 4 | 19.64 | 19.70 | 20.78 | 13.69 |
| | 8 | 39.19 | 39.42 | 40.64 | 27.53 |
| | 16 | 79.02 | 79.50 | 81.22 | 54.66 |
| 512 Kbytes | 2 | 54.9 | 55.0 | 69.5 | 52.3 |
| | 4 | 109.3 | 109.7 | 136.3 | 102.9 |
| | 8 | 217.7 | 219.2 | 262.4 | 206.9 |
| | 16 | 437.1 | 438.1 | 530.7 | 409.6 |

faster than the corresponding message passing implementations.

Another point that can be observed easily is the difference in the timings of the two Converse versions. The difference observed in 64 byte message is due to the overhead of `SelectList()` command used in the 1-receiver version. However for larger messages, this overhead decreases relatively and 1-receiver version becomes the fastest.

## 2.6 Conclusion

In this chapter, we discussed asynchronous message handling approach and described an efficient layer for porting Converse to Parsytec CC distributed memory machine. Although, we ported Converse on top of Parsytec's native message passing layer, the performance results show that Converse performs equally with native message passing on simple (synchronous communication) algorithms and outperforms on algorithms that involve asynchronous communication. Note that no computational task is included in the test programs between communications, which can obviously increase the efficiency of Converse.

Table 2.2: Timings for k-to-all broadcast (for k=n, in msec).

| message size | processor # | $N$-sender $N$-receiver | $N$-sender 1-receiver | PVM | EPX |
|---|---|---|---|---|---|
| 64 bytes | 2 | 1.5 | 1.5 | 1.5 | 0.8 |
| | 4 | 4.8 | 5.5 | 4.7 | 3.1 |
| | 8 | 11.1 | 18.9 | 15.0 | 11.8 |
| | 16 | 24.7 | 48.5 | 33.7 | 25.7 |
| 64 Kbytes | 2 | 8.4 | 8.3 | 10.3 | 8.9 |
| | 4 | 33.1 | 29.7 | 41.4 | 35.4 |
| | 8 | 90.3 | 86.1 | 161.2 | 146.1 |
| | 16 | 255.5 | 235.9 | 403.5 | 369.2 |
| 512 Kbytes | 2 | 48 | 48 | 66 | 65 |
| | 4 | 175 | 161 | 293 | 262 |
| | 8 | 583 | 467 | 1079 | 1114 |
| | 16 | 1726 | 1544 | 2771 | 2797 |

Implementing CMI using a high level message passing library prohibited us to achieve better performance. If the Converse could be implemented on top of hardware using machine primitives and assembler, it would surely perform better.

# Chapter 3

# Hierarchical Radiosity

This section introduces the radiosity approach and hierarchical radiosity algorithm. We present the concept of the approach and discuss existing solutions and algorithms. We also present a sample design for an implementation of a hierarchical radiosity algorithm. While writing this section, we preferred not to deal with details of the problem and implementation since plenty of them can be found in the literature.

## 3.1 Radiosity

Radiosity is a method to produce realistic computer generated images via simulating the distribution of light in an environment. It was introduced to the field of computer graphics by Goral et al. [GTGB84].

The method is mainly based on the idea of heat transfer which is a research area of thermal engineers. Instead of heat transfer, we consider energy transfer between surfaces in the radiosity method. Each surface in a scene absorbs and radiates energy. The energy, which was emitted by the light sources at the beginning, is distributed throughout the scene by energy transfers of surfaces and an equilibrium point is reached. Till reaching the equilibrium point, all surfaces interact with each other and transfer energy. The radiosity method formulates these interactions, builds an equation system simulating balancing of the energy and finds resultant energies for each surface.

Assuming all surfaces have a constant radiosity and reflectance over their own surface, the radiosity equation is formulated as follows for a surface $i$ [GTGB84]:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j \tag{1}$$

where $B_i$ is the radiosity, $E_i$ is the emissivity and $\rho_i$ is the reflectivity of surface $i$, $F_{ij}$ is the form factor between surfaces $i$ and $j$, and $n$ is the number of elements in the scene. This formulation tells us that radiosity of a surface is its self emission plus the radiosity due to all other surfaces' emissions. Since this equation exists for all elements in the scene, they can be combined to obtain a linear system of equations.

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2n} \\ \vdots & \vdots & & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} \tag{2}$$

The values of $\rho_i$ and $E_i$ terms are constant and known in advance. The $F_{ij}$ values are calculated via some techniques explained in the next sections, independent of this equation. The remaining $B_i$ terms are the only unknowns of this equation system.

This equation system may be solved in $O(N^3)$ time with Gaussian elimination method. It is clear that this cannot be an acceptable solution even for simple scenes. Fortunately, the system is diagonally dominant [CW93] and iterative methods such as the Gauss-Seidel or the Jacobi method can be used to solve the system faster. As a property of iterative methods, intermediate solutions can be obtained in early stages of the solution process.

Another major approach to the physical modeling of illumination is ray tracing which is based on the idea of tracing the light rays from eyes of the observer through the scene. In ray tracing, for each pixel on the image plane,

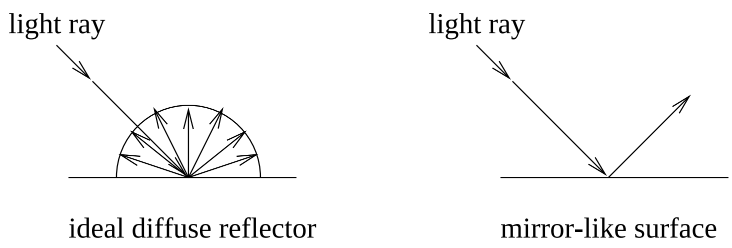


Figure 3.1: Surface types.

a ray is shot and traversed by reflections and refractions among the objects in the scene. It inherently provides point-sampled information which makes it high quality but an expensive method.

All surfaces are assumed to be ideal diffuse reflectors (Lambertian surfaces) in the radiosity method (see Figure 3.1). This restricts its usability, because not all of the real world entities have this type of surfaces. On the other hand, the ray tracing method works solely with mirror-like surfaces. Fortunately, hybrid algorithms ([She94], [SP89]) have been developed which combine these two methods to make it available to process scenes composed of both surface types.

In contrast to view-dependent characteristic of ray-tracing, radiosity is a view-independent technique. Intensities of each surface in the scene are calculated for once at first, and can be used further with any position of camera or viewpoint. This is a very important feature of radiosity which makes preprocessing of radiosities possible. That is, radiosity computations can be done off-line and the results can be used further to render the image using the graphics pipeline.

All parts of the radiosity solution process deal with polygonal surfaces. Therefore, data of the scene to be rendered must be formatted as 3-dimensional coordinates of polygonal surfaces. A surface which is not planar such as sphere must be expressed with smaller polygons. At this point, shading algorithms help us to render non-planar surfaces realistically.

In the next sections, we discuss two fundamental issues of radiosity method, namely form factor and visibility factor calculations. In order to solve the Equation 1 we have to calculate these configuration factors in an efficient way.

Overall performance of radiosity solution process highly depends on the methods we choose for calculation of these factors.

### 3.1.1 Form Factor Calculation

The form factor is the fraction of the energy leaving one patch which lands onto another patch, to the energy leaving the first patch. From a differential area $dA_i$ to a differential area $dA_j$ it is formulated as:

$$F_{didj} = \frac{cos\theta_i cos\theta_j}{\pi r_{ij}^2} dA_j \tag{3}$$

where $\theta_i$ is the angle between the surface normal at $dA_i$ and the vector from $dA_i$ to vector $dA_j$, and $r$ is the distance between the two areas. The terms are illustrated in Figure 3.2. Integrating Equation 3, we get form factor from a finite area $dA_i$ to a differential area $dA_j$:

$$F_{dij} = \int_{A_j} \frac{cos\theta_i cos\theta_j}{\pi r_{ij}^2} dA_j \tag{4}$$

We get the form factor from $A_i$ to $A_j$ by averaging the form factor from $A_i$ to $A_j$ at each point of $A_j$, i.e. we integrate Equation 4 and divide by the area of $A_j$:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\theta_i cos\theta_j}{\pi r_{ij}^2} dA_j dA_i \tag{5}$$

Solving this double area integral analytically is a complex and costly operation [She94]. In the next two sections two numerical methods for computing form factors will be discussed. The hemicube method [SH89] computes the form factors from a differential area to a finite area, whereas the ray-tracing method [WEH89] computes form factors from a finite area to a differential area.

Here are some properties of form factor:

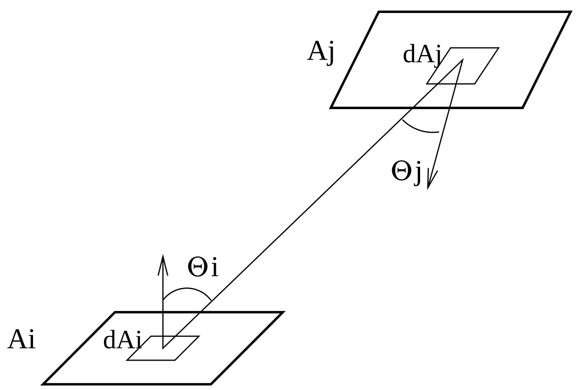


Figure 3.2: Form factor geometry.

- Due to energy conservation rule, sum of the form factors from a patch is equal to unity in an enclosed environment. $\sum_{x=1}^{n} F_{ix} = 1$
- Reciprocity principle $F_{ij} = F_{ji} * A_j/A_i$
- If the surface is convex, $F_{ii} = 0$, i.e. a surface cannot absorb the light it emits, directly.
- Occlusion reduces the form factor of patches. For example, the form factor of invisible patches is 0. The effect of occlusion on form factor is shown in the formula ( $H_{ij}$ is the visibility factor).

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\theta_i cos\theta_j}{\pi r_{ij}^2} H_{ij} dA_j dA_i \qquad (6)$$

### Hemicube Method

In an attempt to calculate form factor, the hemicube method was introduced by Nusselt [SH89]. In this method, a half cube is placed onto the center of the source patch and all other patches are projected onto the faces of this cube. Each pixel is associated with one patch. If two patches are both projected onto the same pixel, the nearest one is associated with the pixel. This operation requires implementation of visibility tests, clipping, projection and z-buffering algorithms.

The form factor associated with a patch projected onto the hemicube is the sum of the form factors at all of the pixels associated with that patch. The resulting form factors calculated with a hemicube give us a row of the radiosity matrix.

Speed of the hemicube computation may be dramatically accelerated if hardware z-buffering is available and scene is completely polygonal. In addition, due to functional similarity of rendering process and hemicube method, existing rendering code can be used for hemicube computation. A disadvantage of this method is that, it suffers from aliasing problems [She94].

#### Ray-tracing Methods

These methods are based on firing sample rays between the two patches and averaging their individual results. Two of them are disc approximation [WEH89] and Monte Carlo integration [CW93] methods. Disc approximation method assumes the source patch as a disc and sufficiently far away from the destination patch. Only one ray is fired, and form factor is calculated. It is a cheap method but fails when the patches are near especially in corners. Analytical extensions are used for these cases. Monte Carlo integration method uses more rays and selects the points of these rays randomly. The result is an approximated value of form factor.

Curved surfaces can also be handled in ray-tracing methods. However the hemicube method is restricted to polygonal objects. Another advantage is that accuracy and speed can be controlled by changing the number of sample points.

### 3.1.2 Visibility Calculation

Visibility factor calculation is one of the most time consuming phases of radiosity process. Whatever the method we choose for the radiosity problem, we cannot avoid calculation of the visibility factor between all surfaces in a scene. This factor has an impact on the form factor value. Elimination of invisible parts of two patches decreases the form factor between them.

Ray tracing techniques are mostly used methods to evaluate the visibility term. We obtain an estimation of the term by firing a number of rays between two patches and counting the rays which do not intersect with another patch. Clearly, accuracy can be adjusted by changing number of the sample rays. We can use common rays with form factor calculation. Note that hemicube method already includes visibility factor calculation inherently and does not need to recalculate this factor.

Some acceleration techniques have been suggested in order to speed up the ray tracing process ([Sam90b], [Sam90a]). These techniques are based on spatial subdivision of environment into cells. Spatial subdivision structures reduce the number of ray-surface intersection tests. The approach which must be followed during subdivision highly depends on the distribution of patches in the environment.

Although it is costly to detect occlusion, it decreases the number of interactions which must be computed. Moreover, by exploiting visibility and spatial coherences, the cost of visibility factor computations can be significantly decreased.

#### Uniform subdivision

This method uniformly subdivides the space containing the environment into a grid of cells. These cells are equal sized and each keeps the list of objects contained in it. The main advantage of this type subdivision is that it lets fast traversal algorithms to be constructed to trace the path of a ray through the grid. Although it is very easy to built and maintain, it is inefficient for most cases. The distribution of objects in the environment must also be uniform to balance the object load of cells. Otherwise, due to unnecessary or insufficient divisions, performance loss can be observed.

#### Octree

Different from uniformly subdivision, this method aims to balance the object load of cells by adaptively subdividing. The whole environment is the root cell of the octree. Each cell is permitted to be subdivided uniformly into 8 cells if it contains more than some number of objects. Subdivision is performed recursively. The resultant structure is a hierarchical tree of octree cells. Pointers to the objects can be kept at the relevant cell or only at the leaf cells.

During constructing octree, it is possible to have large cells that contain only a few small objects. Many rays may enter this region, which do not intersect the object. Although this increases the cost of tracing path of a ray inside the octree, it is still an efficient model. It is easy to build and maintain an octree. Also fast traversal algorithms can be developed to trace path of rays.

The depth of octree is an important parameter for the performance. By changing the maximum number of objects that a cell can contain, the depth can be adjusted to get the highest performance.

#### BSP tree

BSP tree (Binary Space Partitioning tree) method is the most efficient one among the listed methods, which aims to subdivide the space in the most economical way. Each cell is subdivided into 2 subparts by a separator plane each time. The decision of subdivision of a cell is given after finding the correct separator plane. The correctness criterion is leaving same amount of objects at each subparts. The resultant structure is non-uniformly sized cells, each with more or less same number of objects.

Although BSP tree is an efficient method, it tends to be more complicated to build and maintain, in contrast to the other methods. The traversal algorithms are also not as efficient as others. Another disadvantage of BSP tree is dividing the space into 2 parts. The depth of BSP tree may be three times bigger than the octree depth for the same scene.

### 3.1.3 The Ambient Term

In order to avoid getting a dark environment at the early stages of radiosity computation, an ambient term is added to the radiosity of all surfaces. Thus viewing initially dark environments becomes possible. As the radiosity solution converges, this term decreases. Ambient term has no effect on the solution process. It is used only for displaying purposes.

## 3.2 Hierarchical Radiosity

Hierarchical radiosity is a method proposed to solve the radiosity problem [HSA91]. The idea used in hierarchical radiosity is borrowed from the N-body problem. In the N-body problem, each of the $n$ particles in an environment exerts a force on all the other $n$ - 1 particles in that environment, implying $\frac{n(n-1)}{2}$ pairs of interactions. The fast algorithms which can compute all the forces in less than quadratic time are based on two key ideas:

- Numerical calculations are subject to error. Hence, results need only be calculated to within the given precision.
- A group of bodies can be approximated by a single particle if they are positioned sufficiently far away from the body at which their force is being evaluated.

Radiosity and N-body problems share many similarities:

- There are $\frac{n(n-1)}{2}$ pairs of interactions in both problems (without occlusion).
- The fall off factor is $\frac{1}{r^2}$ for both gravitational or electromagnetic forces and the magnitude of the form factor ($r$ is the distance between particles/patches).

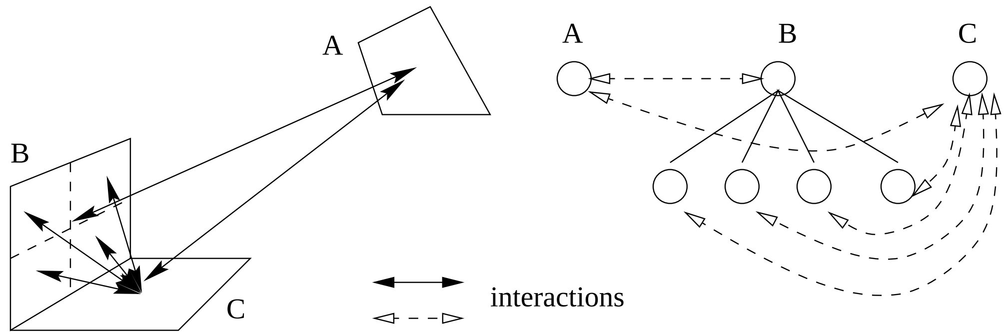


Figure 3.3: Sample hierarchical interactions.

- According to the Newton's third law, gravitational forces are equal and opposite, and, according to the reciprocity principle, form factors between two polygons are related.
- Polygons/bodies can have interaction with each other at different hierarchy levels.

However, certain differences do exist between the two algorithms. One of them is that the force of gravity is not effected by occlusion whereas for visibility calculations of radiosity problem occlusion is an important problem. Another difference exists during the process of hierarchy construction. N-body algorithms group particles together, whereas the radiosity algorithm subdivides initial polygons.

Figure 3.3 represents sample interactions at different hierarchy levels. Patch A is relatively far from patches B and C and so the radiant flux between itself and the other patches is low. Therefore, the error factor in evaluating the radiant flux will be negligible and the patch A is permitted interact with the patches B and C at the coarsest level.

However the patches B and C are near and the radiant flux among them is more than some threshold value. This makes the error factor in evaluating the flux considerably high enough to effect the accuracy of the flux value. In order to decrease the error factor, patch B is divided into subpatches and patch C is permitted to interact with the subpatches instead of B. As the surface area of patch B's subpatches is 1/4 of patch B, the flux will be lower. However, the flux may still be bigger than the threshold. In such cases, we should subdivide

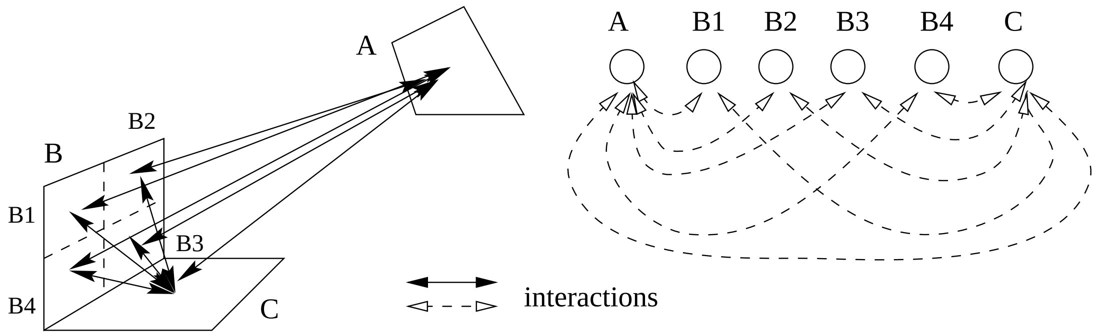


Figure 3.4: Sample interactions of progressive radiosity.

the patches till the error factor becomes trivial.

### 3.2.1 Hierarchical Radiosity vs. Others

As mentioned before, radiosity method has costly operations which make it unusable even for simple scenes. To overcome this problem, different approaches have been proposed. Progressive radiosity and hierarchical radiosity is two of these methods which greatly speed up the computation of radiosity.

Progressive radiosity is based on shooting light from the brightest surface to the environment, iteratively. Shooting is the process of distributing the light energy of a patch out to the rest of the environment. One iteration of progressive radiosity includes choosing the patch with the highest 'unshot' light energy and shooting this energy to the other patches. Since most of the important energy transfers takes place within the first few iterations, useful results can be generated at the early stages of solution process. Also the matrix system converges faster than the conventional solution. Its another advantage is that storage is only $O(N)$ since configuration factors are discarded after they are used.

Figure 3.4 represents patch interactions in the same environment presented in Figure 3.3, which are created according to the progressive radiosity approach. The major drawback of progressive radiosity algorithm is its need of pre-meshed initial geometry. As a rule, patches cannot be subdivided during the solution process. However, the rendering program requires small sized patches which do not carry important amount of energy so as not to cause visual artifacts in the final image. This requires the patches forming the scene to be sufficiently subdivided into subpatches to a very fine resolution in advance. Obviously this will cause too many initial patches and the algorithm will waste time for handling trivial interactions of these trivial patches. Hierarchical radiosity algorithm has been developed just to overcome this problem. A hierarchical algorithm starts with undivided patches and subdivides them dynamically only when necessary. Therefore, hierarchical radiosity does not create or waste time with those trivial interactions. This can be observed in Figure 3.3 and Figure 3.4. Hierarchical radiosity creates 6 interactions whereas progressive radiosity creates 9 interactions. Time and space is saved by ignoring trivial interactions since the accuracy gained by computing them is negligible. As a result of this approach, we can say that hierarchical radiosity works most efficiently for the cases that initial patches are refined into large number of subpatches and less efficiently for the environments with complex initial geometry.

On the other hand, the hierarchical approach has storage problem. Each patch in the scene has to keep its own interaction list including pointers to other patches and configuration factors. This totally makes $O(N^2)$ cost. This is a very prohibitive factor for its usability, because even simple scenes can include more than a thousand patches after subdivisions. Fortunately clustering techniques ([SAG94], [Sil94], [SDS95]) overcome this problem by reducing the number of elements in interaction lists.

Due to its relative simplicity, progressive radiosity is probably the most implemented radiosity algorithm. Hierarchical radiosity is complex and still improving. Studies exist which combine the best sides of these techniques to get better results [HSD94].

## 3.3 Design of An Object-Oriented Hierarchical Radiosity Program

We designed and implemented an object-oriented sequential hierarchical radiosity program in C++ language, and a simple polygon renderer program using OpenGL[1] library. Some details of radiosity implementation are described below.

The program is mainly based on the algorithm presented in the paper [HSA91]. We used Gauss-Seidel iterative approach to solve the radiosity equation (Equation 1). Form factor and visibility factors are calculated using ray-tracing techniques. Different from our reference algorithm, we preferred to implement octree structure to speed up visibility calculations, instead of BSP tree. It is easier and cheaper to implement an octree structure. In addition, hierarchical radiosity deals with relatively small amount of initial data and there is no memory problem for the tree structure. Also the overhead of BSP tree's more complex traversal algorithms might bring extra costs.

This design has been implemented to ease our parallelization study. We tried to develop reusable and independent components as much as possible so as to be able to integrate them to our parallel model without a problem.

The model consists of the following objects and the main program:

- `Vertex` is a point in 3-dimensional space.

  ```
  REALTYPE x,y,z;
  ```

- `Polygon` is a convex planar geometric shape with 3 or 4 vertices.

  ```
  Vertex vertex[4];
  Vertex normal;
  REALTYPE area;
  ```

- `Patch` is a `Polygon` associated with radiosity functions

[1] OpenGL is a registered trademark of Silicon Graphics Inter.

```
Polygon* polygon;
PatchList* interactionList;
REALTYPE B, E, rho;
Patch* parent, *child[4];
```

- `Octree` is a voxel expressed with only 2 vertices.

```
Vertex vertex[2];
Octree* parent, *child[8];
PolygonList* polygonList;
```

- `Algorithm` is the main program that manages the objects and radiosity functions.

The `Vertex`, `Polygon` and `Octree` objects are general purpose objects independent of the radiosity process. `Polygon`s are surfaces of the objects in the environment. The `Patch` object is a `Polygon` associated with attributes and functions related with the radiosity process. These attributes are emissivity, reflectivity, and radiosity values, pointer to interaction list and quadtree pointers. Interaction list is a linked list to keep the pointers to the patches those are fully or partially visible by the list owner patch. Also, relevant information such as configuration factors is stored in this list. Quadtree pointers are used to maintain the hierarchy of patches. Root is always one of the initial patches and the rest of the tree is composed of subpatches of the root patch.

### 3.3.1 Algorithm of Hierarchical Radiosity

The program we built is based on the algorithm given in the paper [HSA91]. Basic steps of a hierarchical radiosity program:

1. **Building environment**

```
read polygons
insert polygons to octree
for all polygons, p
    for all other polygons, q
        if p and q face each other
```

```
            add both polygons to the set of interacting elements
                                                   of each other
```

2. **Solving radiosity equation**

```
while radiosity is not converged
      for each polygon, p
          gather radiance of p from its interacting polygons
```

3. **Rendering polygons**

**gathering radiance, p**

```
for all polygons in p's interaction list, q
    compute form factor and visibility factor of p and q
    if refinement required
       refine the polygon with bigger area
       update interaction lists
    else add the contribution of polygon q to radiosity of p
(/*push and pull the radiosity*/)
if  p is leaf
    add p's emittance to radiosity
else
    gather radiance from all children of p, recursively
    add their radiosities to p's radiosity, with respect to their areas
```

**Solving Radiosity Equation**

As mentioned before, the radiosity equation (Equation 2) can be solved efficiently using iterative methods. In our program, we selected Gauss-Seidel method to implement. An algorithm has been given in the previous section. It is physically equivalent to successively gathering incoming light. In each iteration of the algorithm, radiosity is gathered at each element and pushed down to its children. Once the leaves of the tree are reached, the element's emittance is added, and the radiosities with respect to their areas are passed to upwards. An iteration for an element results by calculating its total radiosity (both gathered and emitted).

### Form Factor Calculation

In the program we used disk approximation method [CCWG88] (Equation 7), which assumes the source polygon as a disk and sufficiently far away from the receiver polygon.

$$F_{ij} = \sum_{k=j_1}^{j_n} \frac{Area_k cos\theta_i cos\theta_k}{\pi r_{ik}^2 + Area_k} \tag{7}$$

where k is an index of the grid cells. See Figure 3.2 for other terms.

In order to increase the accuracy of the factor, both polygons are divided into 4x4 grid and rays are fired from the centers of source subpolygons to the centers of receiver subpolygons randomly. Disk approximation formula is applied to all rays and the result is evaluated after adding them up. We used the rays which are already calculated by the visibility process. That is, the rays which do not intersect with any other polygons are taken into consideration in the form factor calculation. Thus, by merging visibility and form factor calculations as in the hemicube method, process is accelerated and time is saved.

The disk approximation method fails when source patch's area is large relative to the distance. In such cases, either analytical methods are used or the source patch is subdivided till the ratio of area and distance gets normal.

### Visibility Factor Calculation

In order to calculate mutual visibility factor, two visibility tests are applied to each interacting element.

1. Do the polygons face each other? (Only two polygons are considered in this test.)

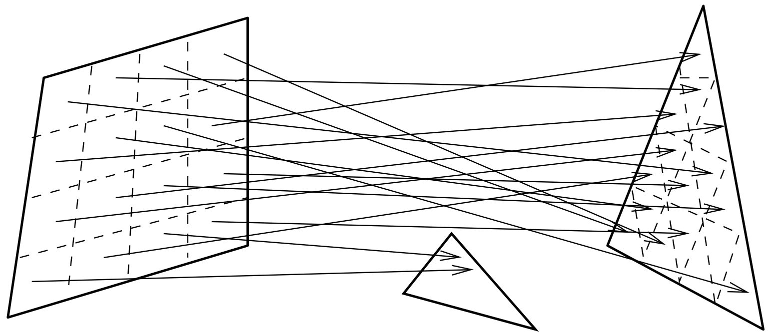

Figure 3.5: Rays fired from a quadrilateral to a triangle to detect occlusion.

2. How much of each polygon is visible from the other polygon given the environment? (All of the other polygons must be considered in this test.)

The first visibility test is achieved by some geometry functions. For the second visibility test, each surface is subdivided into a 2-dimensional 4x4 grid. The grids of both surfaces are matched randomly and rays are fired between them. This is a common method which gives a correct estimation for the visibility factor. If needed, accuracy can be increased by subdividing the surfaces into more cells.

In order to reduce number of the polygons which must be checked for second visibility test, spatial coherence is exploited. The space is subdivided into volumes (voxels) adaptively using an octree structure, until satisfying bounds of some cost function. In our implementation, the cost function is defined as the number of polygons intersecting the volume. An upperbound has been defined also such as 10 polygons per volume.

Polygons are refined during radiosity computations according to an oracle function which is explained in Refinement part of Section 3.3.1. As a result of refinement, new interactions are introduced to the radiosity system which must be processed. Visibility factors are not necessarily computed for every newly created interaction, since we can exploit visibility coherence. If two patches become totally visible, there is no need for further visibility tests between them or between their children. In the same way, if two subpatches become totally invisible relative to each other, then the refinement between them can be immediately terminated. On the other hand, our calculation is based on approximation and if we want to get correct results in all cases, we must give a bound on area and do not inherit visibility factor for big patches.

### Octree

All initial polygons are inserted into the octree at the initialization phase, in order to build the octree structure. Octree is built for only once and used for visibility calculations further. List of pointers to polygons are contained only at the leaf nodes of octree.

While inserting a polygon into the octree, first the smallest voxel that includes the polygon is found quickly. Then, by traversing the subtree of this voxel, link of the polygon is added to all of the leaf voxels which intersect with the polygon. In order to determine whether a voxel intersects with a polygon, four tests are applied one by one:

1. Is there any vertex of the polygon inside the voxel?
   (Yes: intersects, No: next test)
2. Do all of the vertices of the voxel not lay on the same side of the polygon?
   (Yes: next test, No: not intersects)
3. Does any of the edges of the voxel intersect with the polygon?
   (Yes: intersects, No: next test)
4. Does any of the edges of the polygon intersect with the voxel surfaces?
   (Yes: intersects, No: not intersects)

The first two tests are simpler and cheaper than the other tests. For most of the polygons, applying one of the first two tests is enough to get the correct result. The third and the fourth tests are needed infrequently and do not effect the overall performance. After finding a voxel, a cost function is called, which counts the total number of polygons with which the voxel intersects. If the cost function does not permit the insertion of the polygon to the current voxel, the overloaded voxel is subdivided and all of its polygons are transferred to its children. Thus after all, the environment is subdivided adaptively.

In order to use octree voxels in the visibility calculations, an efficient ray-tracing function must be implemented. The strategy we followed in the program is as follows: In order to trace a ray, first of all, we find the voxel which the source vertex is in. For each polygon associated with the voxel, we perform ray-polygon intersection algorithm. If there is an intersection, which means an intervening polygon is encountered, we terminate tracing the current ray. If there is no intersection then we must find the next voxel which the ray is going to visit. After finding the intersection point of ray and voxel, the neighbor voxel which includes this intersection point is computed fastly. Finding nearest common ancestor is a well known method for finding neighbor of a voxel [Sam90b]. This operation is performed until the voxel that the ray ends is visited or encountering an intervening polygon.

A polygon that appears in more than one voxel may be subject to be retested for intersection with the same ray. In order to prevent this, each ray stores a unique tag and this tag is associated with the polygon after the intersection test is performed on it. Therefore, further attempts to test intersection on the same polygons are avoided.

### Refinement

Refinement of a polygon is necessary when the accuracy cannot be satisfied at the current interaction level. In such a case, one of the polygons must be subdivided into subpolygons and radiosity operations must be performed between them. Decision of the patch to be subdivided is given according to the largeness of the surface areas of interacting patches. If patch $A_i$ is larger than $A_j$, then the patch with bigger area, $A_i$ is subdivided and the interaction lists are adapted accordingly. The patch $A_j$ is deleted from the interaction list of $A_i$ and is inserted in the interaction lists of the children of $A_i$.

The refinement based only on the form factors produces unnecessary subdivision for particular orientations? (e.g., around the corners, where the polygons meet and have higher form factors). The BF refinement method [HSA91], which we have chosen to implement for the program, focuses on subdivision of polygons that contain a high level of illumination.

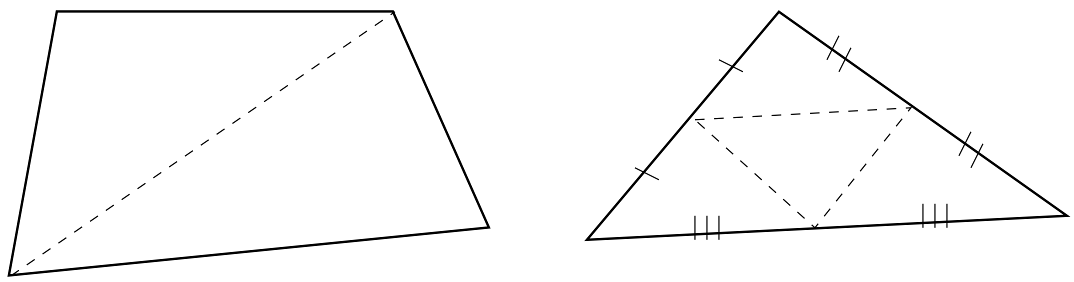

Figure 3.6: Subdivision of a quadrilateral and a triangle.

Polygons with 4 vertices (quadrilaterals) are divided into 2 subpolygons with 3 vertices (triangles), whereas triangles are divided into 4 subtriangles. This is due to the fact that it is more practical to manipulate triangles. The program supports both type of polygons, but after refinement quadrilaterals become triangles. We observed that the geometry algorithms written for quadrilaterals are derived by just performing the algorithms written for triangles twice.

Figure 3.6 shows regular subdivisions of a quadrilateral and a triangle. Although we used regular subdivision for our system, especially subdivisions along shadow boundaries require irregular subdivisions for a better result. This is a trade-off between time and quality.

### Push-and-Pull Phase

As shown in the algorithm, after finishing gathering radiance operation of a patch and all of its subpatches, we push and pull the radiosities of those patches. This operation is a redistribution of radiosity collected by different hierarchy levels of the patch.

Push and pull phase consists of top-down and bottom-up traversals for each quadtree. In the top-down traversal, the contributions of parent nodes are transferred to their children. In bottom-up traversal, conversely, the contributions of children are transferred to their parents with respect to their area. As a result, consistency of radiance values of patches at different hierarchy levels is preserved.

#### Multigridding

Multigridding is a technique to speed up the convergence of radiosity solution system by letting the interactions remain at coarser levels as much as possible. This is incorporated into the algorithm by starting with a high error tolerance for interaction refinements and decreasing it at some rate after each iteration. Therefore, first iterations which finer level interactions do not necessarily have to exist, can be performed cheaply.

## 3.4 Further Improvements on Radiosity Process

Many improvements have been introduced to the computer society in the last decade, such as lazy linking [HSD94], clustering [SAG94] [Sil94] [SDS95], bidirectional radiosity [DBSW97], importance-driven model [SAS92], discontinuity meshes [LTG92] [LTG93], hybrid algorithms [She94] [SP89].

Lazy linking is a strategy to reduce the initial linking cost by delaying the creation of insignificant links. The effect of this ignored interactions is then approximated using an ambient light term when the solution is displayed. This method enables a radiosity solution to be calculated in a shorter amount of time.

Clustering is a method to group nearby polygons into object hierarchies for the purpose of evaluating their energy exchanges with distant objects. In this method, instead of many trivial interactions, sufficiently distant objects perform group interaction. It does not only speed up the computations but decreases the storage requirements as well. As a result, simulating the radiosity of very complex scenes becomes possible.

Importance-driven radiosity focuses computation on the parts of a scene that has more effect on a particular image. It is only applicable to the wavelet radiosity methods.

Discontinuity meshing attempts to model shadows accurately by creating meshes at shadow boundaries. Although the method eliminates visual artifacts caused by the shadow boundaries, it is computationally expensive and complex to implement.

Hybrid algorithms have been developed to calculate the illumination of environments consisting both specular and diffuse type of objects.

# Chapter 4

# Parallelization of Hierarchical Radiosity

## 4.1 Introduction

The motivation behind this work is to investigate the feasibility of parallel processing so as to produce a fast radiosity solver. High speed and low storage are the only two things that radiosity problem is in need of. Exploiting parallelism is one of the practical ways of addressing these problems.

This chapter presents detailed information about our parallelization study of hierarchical radiosity algorithm. First, we provide an overview of parallelism in hierarchical radiosity method. Then, we explain our design and implementation issues. In the last section, we discuss some issues related with high performance such as load balancing.

## 4.2 Characteristics of Radiosity Data

Inputs of radiosity process are the polygons which represent the surfaces of objects in a scene. All the objects are assumed to have diffuse reflector surfaces. In our algorithm, we used polygons with 3 or 4 vertices.

The objective of hierarchical radiosity algorithm is to evaluate the interactions of polygons in terms of light energy and calculate final intensity of each polygon. Two patches are said to be interacting if they are completely or partially visible to each other. Such patches must keep information about each other so as to evaluate this interaction. This makes the patches dependent to each other. Each patch maintains a list of pointers to its interacting patches. This dependence introduces new difficulties to the parallelization studies.

As an important property of the algorithm, the patches are subdivided during the solution process. Evaluation of an interaction may result in such a subdivision. Refinement of an interaction is applied when the radiant flux between the patches is big enough to effect the accuracy of resultant radiosity. After refinement, the old interaction is removed and new interactions from the undivided patch to the subpatches of the divided patch, which must also be evaluated, are established. As a result, the data to be processed dynamically grows and achieving good load balance becomes more difficult. On the other hand, progressive radiosity deals with only static data. Instead of dynamically subdividing, in progressive radiosity, patches are subdivided in advance. However, it suffers from overhead of manipulating big amount of data.

The data used in hierarchical radiosity is not meshed in advance, in contrast to progressive radiosity. Because, hierarchical radiosity approach works most efficiently for the cases that initial patches are refined into large number of subpatches. Such scenes produce very few interactions relative to the progressive radiosity. Also, it is less efficient for the environments with complex initial geometry. The overhead of hierarchical approach may cause bad performance for such environments.

## 4.3 Previous Work

Parallelization of hierarchical radiosity on distributed memory computers is not easy due to its prohibitive characteristics of input data, as explained in the previous section. There is no efficient methods that been developed so far in this field. Most of the existing studies are on shared memory machines which are generally based on management of a task pool [PRR96], [VC95], [RS97], [SHT+95].

Richard and Singh state the difficulty of predicting the amount of work associated with a patch or interaction in their paper [RS97]. They use distributed task queues and permit task stealing to achieve load balance among processors dynamically. Their study also involves specular radiosity in addition to diffuse one. Specular radiosity changes some characteristics of the program such as high percentage of the visibility calculations. While the diffuse program spends 90% of its execution time computing visibility, the specular+diffuse program spends 68% of its execution time gathering specular radiosity and 6% computing visibility. Their speed up of the specular+diffuse program is 26.3 on 32 processors. They note that the major reason for the loss in speed up is the synchronization overhead.

Singh et al. presented a parallel hierarchical radiosity algorithm on a cache-coherent shared address space multiprocessor [SHT+95]. They use a distributed task-queuing mechanism to reduce contentions. Therefore, a cost estimation function is needed to partition the tasks among processors. Task stealing is allowed to preserve balance of processor loads. The gained speed up is almost same as the study in [RS97].

Distributed memory implementations have not produced satisfactory results as others so far. This comment is explicitly stated by most of the researchers studying on this subject. Moreover in [SGL94], it is mentioned that they abandoned their parallelization project because of not being worthwhile, after getting 11-fold speed ups on 32 processors.

Garmann et al. presented a parallelization study on CM-5, which is a MIMD-type multicomputer [GBM94]. Their speed up shows the difficulty of parallel processing for hierarchical radiosity algorithm, only 8.4 on 64 processors. Garmann's approach to the algorithm was manipulation of a huge graph.

In [FY97], Feng and Yang claim that they have developed an efficient implementation. Their study is based on CV-sets for visibility computations. The scene is partitioned into cells and all polygons visible by a cell are kept in the

CV-set of that cell. This notion helps to assign the scene data to processors with a good initial load balance. Initial distribution is done by estimating the load of patches. His load estimation function depends only on the area of patches. As we discussed in Section 5.2 patch area is not sufficient for an accurate load estimation. They presented performance results of complex (large) scenes up to 8 processors. However their input scenes are composed of fairly independent subscenes which are slightly interacting with each other. Therefore CV-set approach assigns each subscene to a different processor which results in good speed ups. However, CV-set approach will not handle dynamically changing load balance for arbitrary input scenes.

On the other hand, progressive radiosity is more suitable for parallelization because of its static data. Many studies exist which include efficient methods [SWPW95], [CAO93], [ACO96], [SSV95], [GRS95]. Most of the solutions are based on parallel ray tracing for the computation of the form factors and visibility rates. The distribution algorithm is straightforward, each processor gets an equal number of patches. Since data is constant, static load balancing produces very efficient solutions. Parallelization of gathering radiosity is also investigated in the literature [KAO97].

## 4.4 The Underlying System

The implementation has been done on Charm++ system. Charm++ is a parallel object-oriented programming language. Charm++ allows us to define parallel objects (called chares) and it supports dynamic creation of these parallel objects. Different from the conventional message-passing style, Charm++ provides message-driven execution. Methods of remote parallel objects can be invoked asynchronously. That is, whenever a message arrives for an object, the method specified by the message is scheduled for execution immediately. Hence, the message-driven execution helps to overlap idle times of processors by executing methods in any order. More information about Charm++ and message-driven execution can be found in [CHRM97] and [KG95].

In our implementation, we particularly made extensive use of "branched office chares" (BOCs) of Charm++ language. A BOC is a parallel object replicated at every processor (branches). One of the usages of BOC's is to perform collective operations effectively. Each branch can handle local operations in a processor and then branches communicate with each other to complete the collective operation.

Another important property of Charm++ language is its portability. We had chances to run our program on different machines such as network of workstations and Parsytec. The performance results are obtained from Parsytec machine which has 24 processing nodes. It is a MIMD machine and has a high-speed bandwidths among nodes. More information can be found in Section 2.3 and manual [EPX95].

## 4.5 Design

The radiosity algorithm is based on the original hierarchical radiosity algorithm presented in the paper [HSA91]. We have also presented the sequential version of this algorithm in the previous chapter. Parallel version has been developed over this sequential version.

Developing parallel programs always requires a careful analysis of data and task flow. As it is mentioned earlier, hierarchical radiosity is an iterative algorithm. It has strong computational and data dependencies both within an iteration and between successive iterations. Parallelism within an iteration should be investigated individually while considering the dependencies between successive iterations. The conventional parallelization approach for MIMD computers creates tasks with minimum dependencies and distributes them to processors equally as much as possible. The processors execute these tasks concurrently with local information. Inevitably, the dependencies of tasks cause the processors to synchronize with each other and transfer data. The processors also follow either static or dynamic strategies in order to preserve their load balance.

In this section, we will discuss the following design issues:

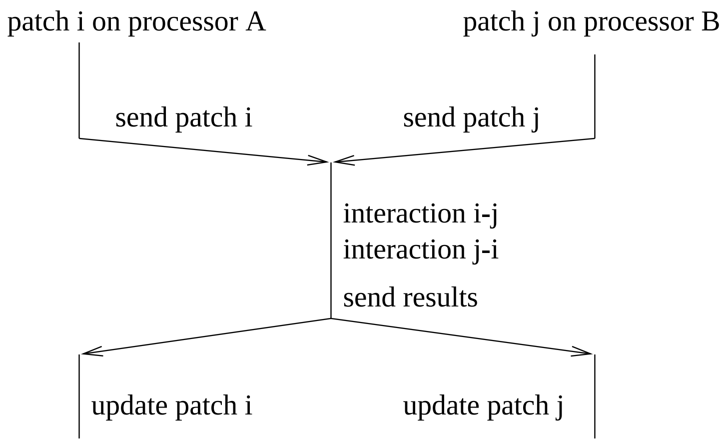


Figure 4.1: Evaluating an interaction on a processor which does not own any of the interacting patches.

- how we partition patches across processors and map computations (patch interactions) to processors,
- how we deal with communication,
- how we deal with the dynamically changing load balance as the patches get refined, and
- how we hide the difficulties of parallelization and communication from the computational algorithm.

In the radiosity process, the unit work is the evaluation of an interaction which is established between any two mutually visible patches. In order to evaluate an interaction on a processor, we need the data of the interacting patches. After completing the interaction, the data of both patches need to be updated.

We can map an interaction to any processor. However, if its interacting patches are not owned by the local processor, then, the interaction must request its data from owner processors. When data arrives, interaction is evaluated and the results are sent back to owner processors. To reduce the communication, we need to map interactions to processors which own the patches. As illustrated in Figure 4.1, mapping an interaction to a different processor (which does not own any of the interacting patches) might be necessitated due to load balancing requirement. Therefore, it is still an option in the design phase.

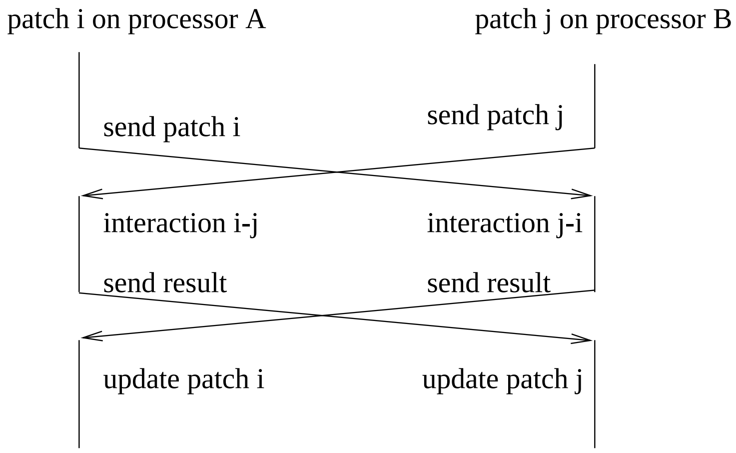


Figure 4.2: Evaluating an interaction on both of the processors which own the interacting patches.

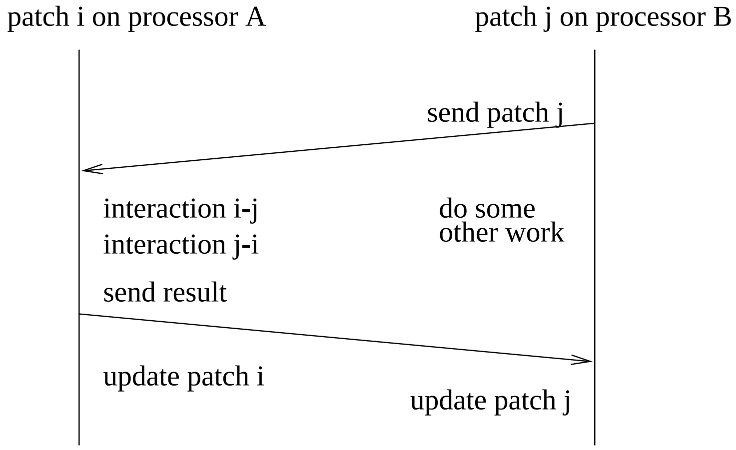


Figure 4.3: Evaluating an interaction on one of the processor which owns any of the interacting patches.

We can map interactions to the processors which own the interacting patches. If both patches of an interaction are on the same processor, then we can map the interaction to that processor. However, if one of the patches is on a different processor, we execute interactions ($i$-$j$ and $j$-$i$ interactions) on both processors. For patch-$i$ and patch-$j$, we can map interaction $i$-$j$ to processor-patch-$i$ (processor that owns patch-$i$) and interaction $j$-$i$ to processor-patch-$j$ as shown in Figure 4.2. So it requires total 4 messages and each processor can evaluate interaction $i$-$j$ and $j$-$i$ in parallel.

If we map both interactions $i$-$j$ and $j$-$i$ to one processor, either processor-patch-$i$ or processor-patch-$j$, then the number of messages will be two and

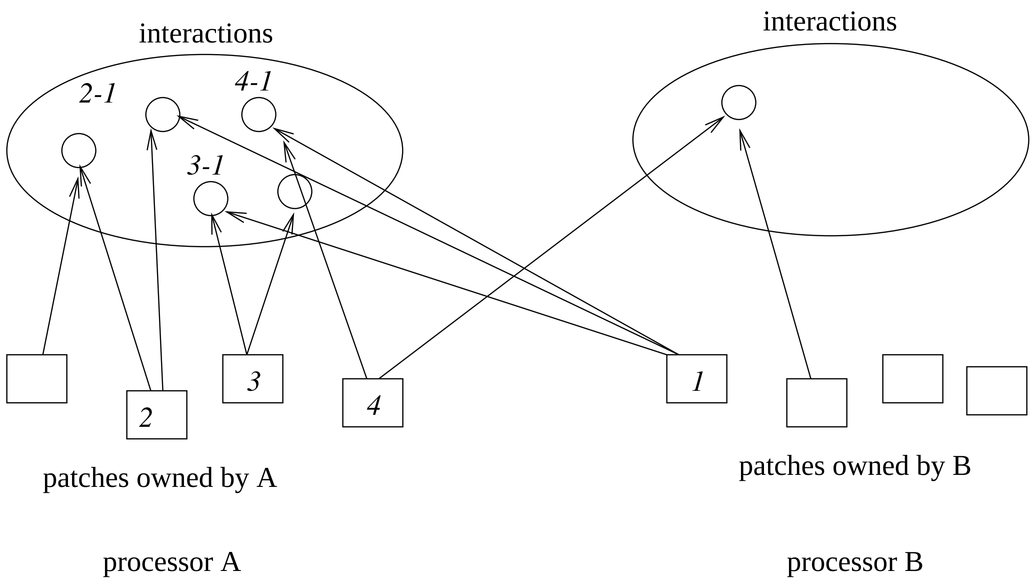


Figure 4.4: Interactions across processors.

both interactions will be evaluated on one processor one after the other (see Figure 4.3).

Since there are many interactions in the radiosity calculations that can keep processors busy, we decided to map interactions $i$-$j$ and $j$-$i$ to one processor (i.e, evaluating interactions $i$-$j$ and $j$-$i$ on the same processor will still provide enough parallelism) to reduce communication cost. But, how will processor-patch-$i$ and processor-patch-$j$ decide which one will evaluate the interactions? Making this decision dynamically is again attractive from load balancing point of view, however, it will cause communication between processors. Instead, a static decision scheme which eliminates communication has been used. Both interactions between patch-$i$ and patch-$j$ are assigned to the processor who owns the patch with larger area. The rationale behind this is that, the larger patch will most likely be divided into smaller subpatches and interactions can be pushed down more easily. This will be clear, later, as we discuss the parallel algorithm.

Patches are partitioned across processors and interactions are mapped to processors according the formula above. Later, in Section 4.6.1, we will discuss how patches are assigned to processors in order to balance the computational load and reduce communication. Before that we want to look at the communication patterns and improve the design with techniques orthogonal to

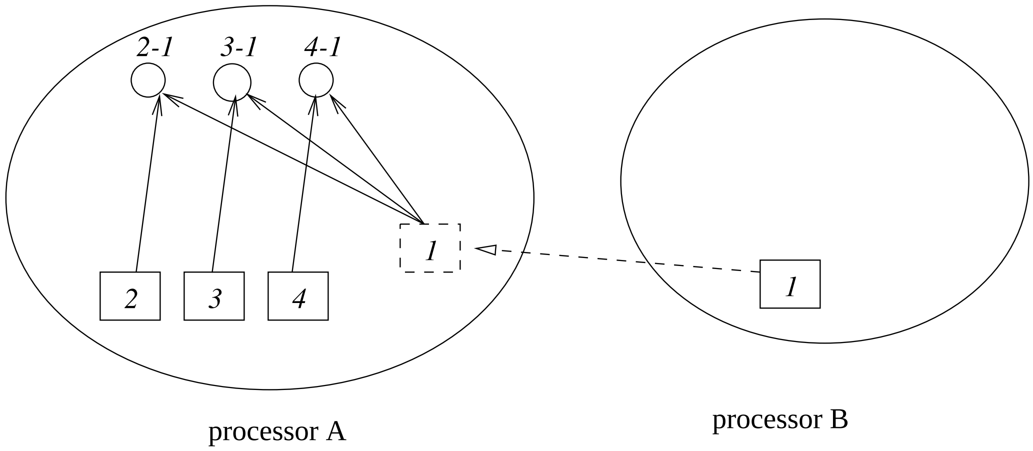


Figure 4.5: Interactions using proxy patches.

mapping.

Since the interaction pattern is quite irregular, whatever the mapping is, there will always be many across-processor interactions. If we design our software such that each interaction object responsible for gathering the data that it needs, this will result in multiple messages for the same patch data. As can be seen in Figure 4.4, for example, the data of patch-*1* from processor-B is sent to interactions *2-1*, *3-1* and *4-1*.

What we need is to send the patch-*1* data in a single message and allow interactions *2-1*, *3-1* and *4-1* use the same data. Interaction objects can do this by communicating with each other, and figure out which ones need the same data etc. But this will result in inefficient and complex code for interaction objects. In order to simplify the design of interaction calculations (i.e., free them from parallelization issues and let them do only calculations) and also reduce the number of messages, we propose to represent patches in remote processors by a special object called proxy patches [KSB+98]. A proxy patch is a representative of a patch at a different processor. We keep the incoming guest patches' data in a proxy structure. In Figure 4.5 we can see a sample usage of proxy patch. To evaluate interactions *2-1*, *3-1* and *4-1*, processor B sends the proxy of patch-*1* to processor A. Here are the advantages of using such a structure:

- Proxies provide that there is at most one representative of each patch in a processor. There may be too many local patches in a processor requesting the same remote patch to evaluate their interaction. Instead of fetching the remote patch at each time when a local patch requests it, the remote patch is fetched once and used for further requests. Also instead of updating proxy's original patch after every evaluation, this is done at the end for only once.

- Proxies provide the radiosity functions to operate as if all of the patches are local. A proxy patch carry enough information to be treated as a local patch by radiosity functions. That is, the radiosity functions execute the same instructions for patches without considering whether it is proxy or not. This flexibility greatly removes complexity of radiosity functions.

- In our design, proxies are not requested since they know where to go. Each patch knows from which processors it is going to be requested in advance and sends its proxy to these processors without waiting a request. Proxies go and replace into these processors and wait to be processed. By skipping the request phase, the overhead of sending/receiving proxies is greatly minimized.

- Using proxies also decreases volume of communication between processors. Static parts of the proxy data are transferred only once to the relevant processors at its first visit. These data occupy the reserved memory area of a proxy and used within the rest of the solution process. As a result, while sending proxy data, we transfer only its updated data fields, such as radiance.

Without proxies, the code will be much more complicated. In order to evaluate an interaction of a patch, its processor would have to send a request to the owner processor of its interacting patch. This method has been implemented in Garmann's study [GBM94] where the speed up is very low. It is clear that many patches will be remote as the processor number increases. If we send requests for every remote patches, we waste too much time for communication and cannot get scalable timings.

If we let each patch to update its proxies, then there will be communication for each of them. Instead, all the updates of proxies between any two processors

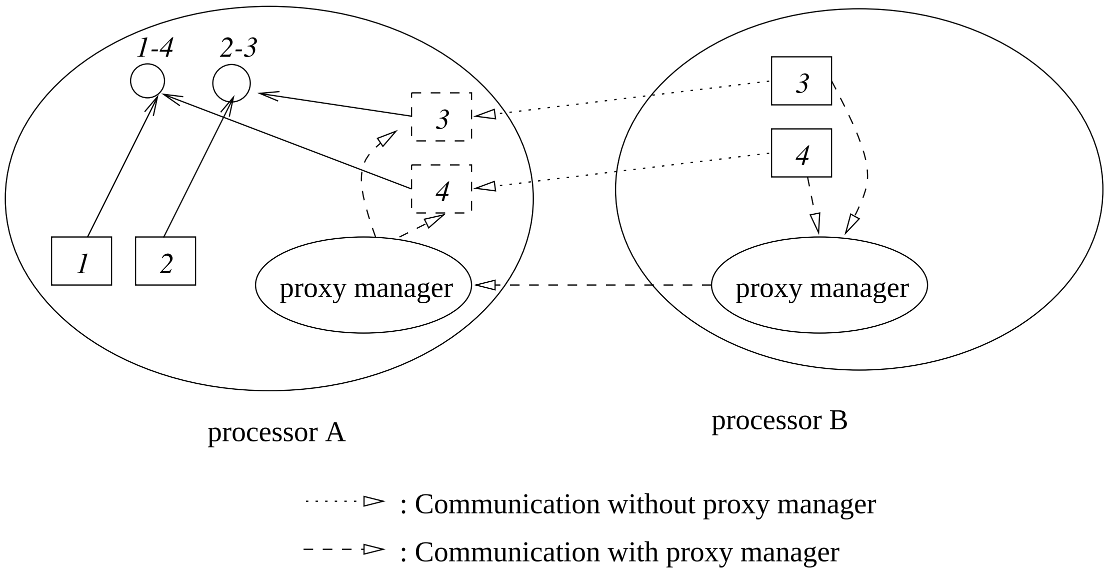


Figure 4.6: Interactions using proxy manager.

can be handled together. This requires a communication mechanism which gathers the data of all patches on a processor and distributes them to only the processors that need them. We can use BOC parallel objects of the Charm++ language to implement this communication mechanism easily. This branched object (proxy manager) deals with update of proxy patches by establishing communication with other processors' proxy managers.

The use of proxy manager is illustrated in Figure 4.6. Instead of individual communication of patch objects, proxy manager collects all these requests and perform only one communication.

### 4.5.1 Dynamic Load Balancing and Patch Migration

Evaluation of interactions causes changes in the structure of patches, such as removing pointers from interaction list, adding new pointers to interaction list, subdivision, etc. Subdivision of patches increases the amount of interactions that should be evaluated. This means the load of such processors increases proportional to the rate of subdivision, resulting load imbalance between processors. This load imbalance is inevitable since we do not know which patch will subdivide how many times in advance. In such cases, a load balancing algorithm must deal with this problem by rebalancing the load during the execution. In order to do this, we should be able to detect load imbalance and move some patches from highly loaded processors to less loaded ones. In Section 4.6.4 we explain the design and implementation of the patch migration algorithm in detail.

### 4.5.2 Subdivision Depth Limit

Different from the sequential version, we had to restrict subdivision of patches within an iteration. As we have said, in order to evaluate an interaction, it is required that the interacting patches' data are locally present. If we do not limit subdivisions, then we have to transfer proxy patches together with all of its existing subpatches. This will obviously increase communication volume very much. Also, since we cannot guarantee that an interaction will refine within the current iteration, it would be useless to send all of those data in most cases. To get a reasonable solution, we transfer proxy patches together with their subtree up to a level and restrict these patches to be subdivided at most this limit times.

### 4.5.3 Visibility Calculation

The input geometry of hierarchical radiosity, in contrast to progressive radiosity, consists of unmeshed polygons. That is, it has much smaller number of polygons than the final geometry has. Therefore, we can replicate the initial input geometry on every processor to perform the visibility calculations. This does not bring a significant overhead.

However, other unhierarchical parallel radiosity approaches suffer from global visibility calculations. Since the input geometry is quite large and distributed to processors, polygon-polygon visibility tests cannot be performed locally. For most of the ray-polygon intersection tests, the processors require to establish communication, which obviously decreases the overall performance.

### 4.5.4 Message-Driven Execution

Our scheme also employs message-driven execution. In message-driven style, each processor manipulates a bunch of objects or processes. These objects or processes communicate with each other via sending messages. There is a message pool which is maintained by the scheduler of the processor for this aim. The scheduler is a utility of Charm++/Converse system which provides message-driven execution model and ability to invoke methods of remote C++ objects. Whenever the processor is idle, the scheduler picks up a message from the pool and invokes the requested object with the message.

### 4.5.5 Algorithm

Algorithm roughly involves the following steps:

```
1. initial linking & distribution

2. until convergence

   (a) send patch data, update proxy data
       perform radiosity computation

   (b) send proxy data, update patch data

   (c) push & pull phase

   (d) broadcast radiosity and load information
       if not converged, balance load

       i. if required, perform migration
```

As we have said, each patch is assigned to and located in only one processor. In order to provide all the patches locally present during the computation phase, processors send data of their patches and update data of other processors' proxy patches. The updated proxy patches are then posted to their owner processors to preserve their consistency. This is the reason of sending and receiving data of patches at the beginning and end of each iteration. Transferring data of patches and proxy patches are undertaken by proxy manager, whereas computational tasks are handled by patch manager.

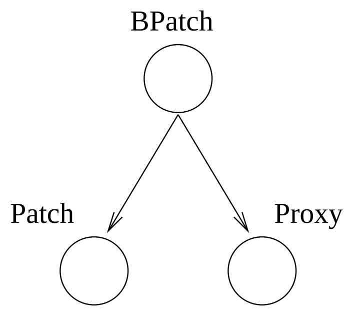


Figure 4.7: Base and inherited classes for patches and proxies.

Details of algorithm and a work flow schema are presented in the next sections.

### 4.5.6 Object Oriented Design

The model is derived from the design of the sequential algorithm. `Vertex`, `Polygon`, `Octnode` and `Octree` objects are just the same objects of the sequential version (see Section 3.3). Different from the sequential version, we are forced to define different classes for local patches and proxy patches. Although they have almost same data fields, local patches and proxy patches should be handled differently on some events (subdivision, indexing etc.). We defined a base class `BPatch` for these classes (see Figure 4.7). We put similar things into this class, and specific things into their own class. Flexibility of using base class greatly removed complexity of using different classes. `PatchManager` and `ProxyManager` classes are the branched office chares created by the Charm system one in each processor, which are charged for management of local and proxy patches.

- `BPatch`: Base class for `Patch` and `Proxy` objects.

```
Polygon* polygon;
PatchList* migratedInteractionsList;
REALTYPE oldB, newB, rho;
Patch* parent, *child[4];
int globalId, localId;
int* localIds;
```

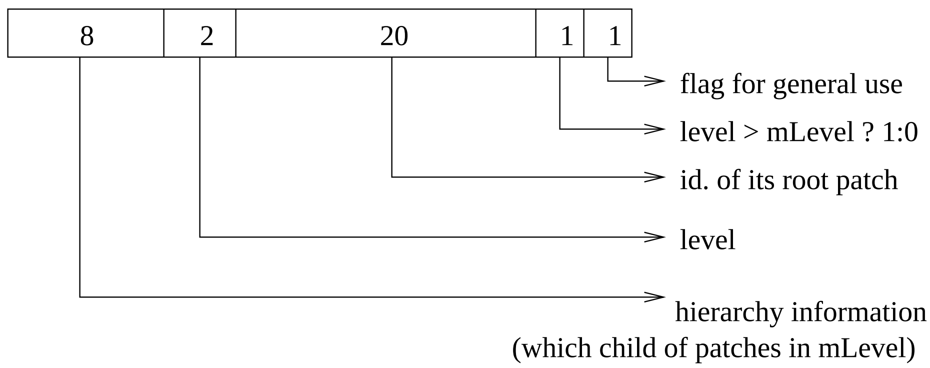


Figure 4.8: Parts of a global id. consisted of 4 bytes.

Different from the sequential version, `BPatch` class has local/global id.s and a new interaction list. Global id. (`globalId`) is used to keep hierarchy information for subpatches, up to some level. Its parts can be seen in the Figure 4.8 (mLevel is the maximum level of a patch selected to migrate). Global id. is also used to determine the owner processor of the patch. Local id.s (`localId, localIds`) are used for direct accesses. Every patch keeps information of its local id.s at all processors. The new interaction list, `migratedInteractionsList`, is used in case of migration to keep track of the interactions that are moved together with the migrated patch.

– `Patch`: This is the class of local patches.

```
REALTYPE E;
PatchList* interactionList;
int interactingProcs;
```

Besides the inherited data from `BPatch` class, `Patch` class keeps an interaction list, emissivity and interacting processors information. Interaction list is same as the one used in sequential version, and only present for `Patch` class. `interactingProcs` is an integer but used as a bit-array. As we have mentioned before, a patch knows from which processors it will be requested in advance and sends proxies to there. `interactingProcs` is used to keep these processors.

– `Proxy`: This is the class of proxies of remote patches.

```
int homeId;
int LUIN;
```

A proxy keeps information about its local id. (`homeId`) at its home processor. Note that, `localIds` array is null for proxies unless there is

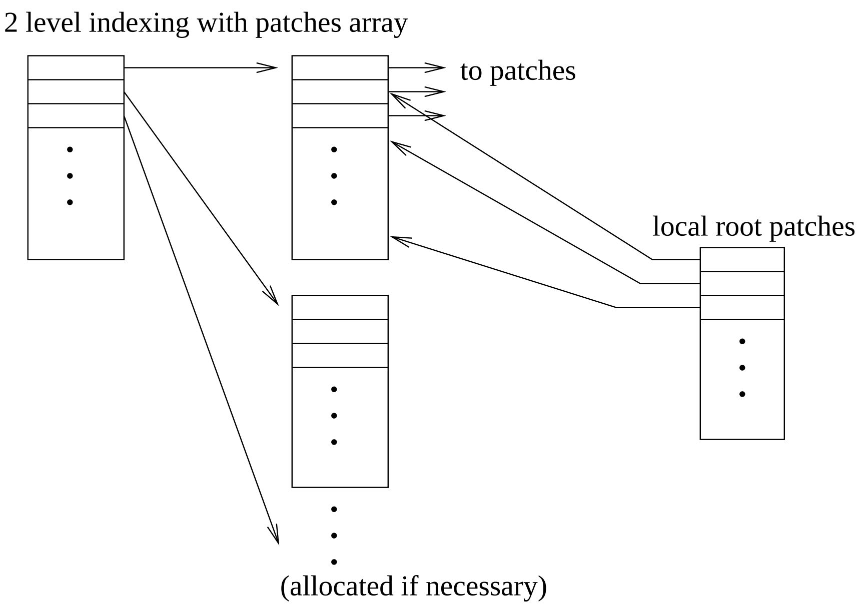


Figure 4.9: Indexing strategy of all of the existing patches.

migration in the current processor. `LUIN` is the last iteration number that the proxy data is updated. This is used to understand whether the proxy data should be sent to its home or not at the end of iterations. If its data is not modified in the current iteration, we do not need to spend time to send it. It obviously decreases the volume of communication.

- `PatchManager`: As its name implies, this class is responsible to manage the patch data in each processor. This task involves performing necessary radiosity calculations on this class. As a result of solving radiosity system, patch intensities are calculated and patches become ready to be rendered. `PatchManager` works in coordination with `ProxyManager`. During its execution, `PatchManager` deals with only computational part of algorithm.

```
BPatch* patches[][];
BPatch* localRootPatches[];
int totalPatch;
Octree* octree;
REALTYPE newRad, oldRad;
int loadTransferMatrix[MAXPROC][MAXPROC];
int mLevel, pLevel; int iterationNo; ...
```

Patches are stored in the memory locations allocated by `PatchManager`. Their indexing is also done by this manager. It is not possible to allocate a fixed size of memory for patches since they dynamically increase during execution. Dynamic memory allocation can prevent indexing patches for accessing them directly. We built a 2-level indexing strategy, by which direct accessing is provided and memory is used economically. The first level is the master level and points to the second level index arrays. The second level index arrays point to patches and are allocated when necessary. Figure 4.9 illustrates how this indexing is achieved. `localRootPatches` array points to root patches that are local in current processor. Another method for accessing patches might be based on hashing. Although it is easy to implement and removes all complexity of indexing, it is not a measurable method. We have too many accesses to the patches during process and not so patient to tolerate the overhead of hashing.

Load transfer matrix keeps track of migration operation: from which processor to which, and how much load must be migrated. `mLevel` is the maximum level of a patch selected to migrate. Proxies come/go with at most `pLevel` level subtree data.

- `ProxyManager`: `ProxyManager` is responsible to manage proxies in each processor. It establishes communication with other processors and exchanges proxy information. Such requests are made by `PatchManager`. Data of proxies are packed to send and unpacked when received.

```
char* queue[];
int queueSize, queueCounter; ...
```

  Queues are used to store messages that are going to be sent. These messages are the proxies which have to be packed into a contiguous memory area in order to be sent. In a processor, each queue is associated with a separate processor.

- `Main`: It provides necessary information to the `PatchManager`'s of each processor to start the radiosity process. The `Main` object is single for all processors.

### 4.5.7 Flow of the Algorithm

The program starts with the execution of the main object which is single for all processors. The main object reads input geometry from disk and broadcasts this information to all processors' patch manager. Then it waits for the solution to render the image, in sleep mode.

Receiving the patch data, patch managers do some necessary operations to start gathering radiance process. First the octree is constructed with the input data. Since it is a fast and low storage operation, all the processors do the same things and construct the same octree. Then we need to establish links between interacting patches. To do this, we perform visibility test to all of the possible patch pairs and create links among the patches which are visible to each other. We give the ownership of the link to the patch with bigger area than its interacting patch. The aim is to increase the possibility of providing the patches to subdivide locally. Interactions are processed on the processor of the patch which has the interaction. If the interaction is to be subdivided, we always choose the bigger one which we also want to be local, to subdivide. Dividing proxy patches increases volume of communication between processors.

After building interaction lists, we require to assign the input patches to processors as a rule of parallel processing. If we aim to assign equal loads, we have to develop a mechanism to estimate loads of patches. To do this, we execute one iteration of radiosity algorithm locally and without refinement to see the properties of patches' interaction lists in a little bit more detail. Since we neglect the refinement part, it does not take too much time. After finishing this step, we choose the input patches in a manner that we discuss in the load balancing section (e.g. random, input order) and assign them to processors by equalizing their loads.

Which patch belongs to which processor information is kept on a table instead of on patches. Migration operation changes processors of patches. In such a case, updating the table is enough to indicate this change. Since migration is permitted for patches up to some level, table keeps information only for that much level. Therefore we not only gain from time but spend less memory with this table structure as well.

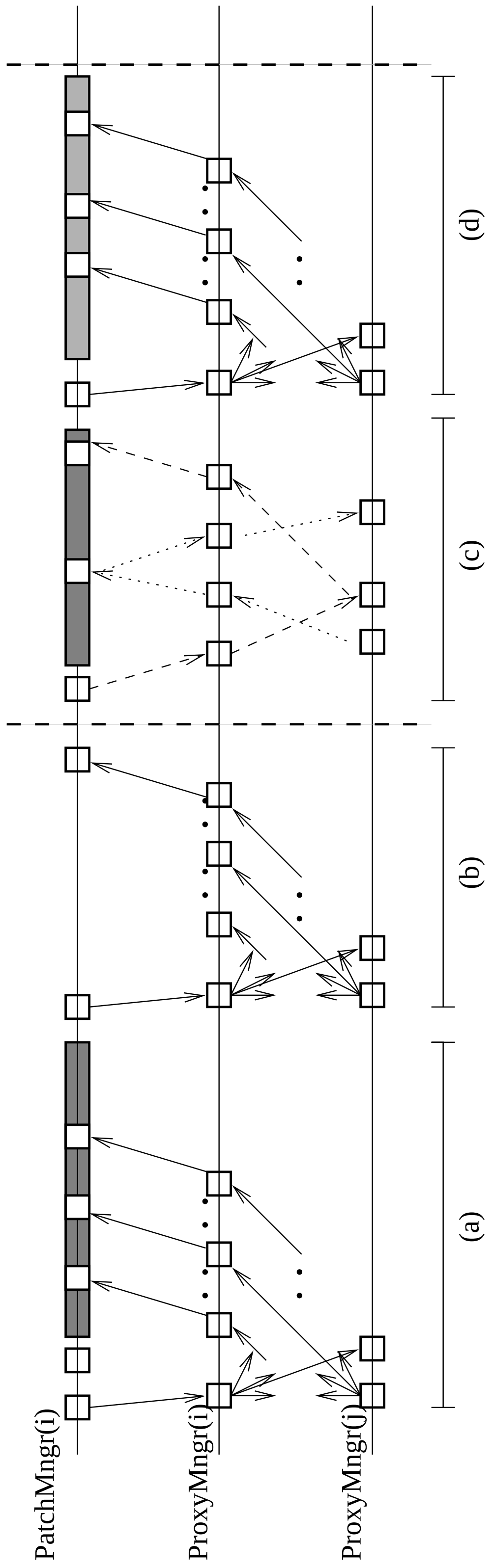


Figure 4.10: Work flow in an iteration (without migration).

After assigning the patches to processors using a load balancing technique, we start to perform radiosity computations. The work flow is simulated in Figure 4.10. Four basic parts take place in an iteration.

- **step (a): Send data of patches - Update data of proxy patches**

  In order to evaluate an interaction, processor of its owner patch must provide the other patch to be present locally. Since each patch is assigned to and located in only one processor, the processor of the owner patch must communicate and fetch other patch's data to perform the interaction evaluation operation. This step involves this communication. For the evaluation of all interactions, each processor sends its local patches to other processors which need them. Since each patch knows which processors need its data, there is no a 'request' ing patch phase. Receiving the patches, the processors start to perform radiosity computations on them.

  As illustrated in Figure 4.10-a, patch manager triggers proxy manager to send the patch data, with a message. Then proxy manager establishes communication with other processors and transfers the local patch data. While trying to send the data, the proxy manager may of course receive data posted from other processors for the same aim. In such a case, patch manager detects interactions related with the coming data and creates tasks to evaluate them.

  The sent data of a patch must be sufficient for the operations that will take place in current iteration. By taking into account that interactions may refine, we have to send patches with their subtree data up to a level (see Section 4.5.2). We have called this level limit as `pLevel` in the previous section. Thus while sending a patch's data, we pack its subpatches' data up to a level and send it. Packing is required to get a contiguous memory area. This is handled by proxy managers. The receiver proxy manager unpacks the message and update data of proxies accordingly.

- **step (a): Evaluate interactions**

  This step is executed concurrently with the previous step. While patch managers evaluate existing interactions, proxy managers send/receive data of patch/proxy if necessary. Communication and computation are overlapped here.

Interaction evaluation function is an entry function of patch manager. That is, in order to request evaluation of an interaction, a message must be sent to patch manager. There is a queue for messages waiting to be processed. The sent message is added to this queue and processed when its order comes. This queue is managed by the scheduler of Charm system. All of the interaction evaluation operations in this step are requested by sending messages to the patch manager.

While evaluating interactions, proxy patches and local patches are considered same. Radiosity functions accept both of them since they have enough information provided by patch and proxy managers. However, in case of migration, we may fail to subdivide some patches. This case is explained in further sections.

As discussed in Section 4.5.2, there is a limit on the subdivision depth of patches. Therefore, while evaluating interactions, it is possible that some interactions cannot be refined within the current iteration.

- **step (b): Send data of necessary proxies to update - Update data of my patches**

  After evaluating all the interactions related with the proxy patches, their updated data have to be sent to their home processors. Thus the gathered radiosity of all proxy patches are contributed to their original patch. Also hierarchy information is updated for patches that are subdivided at remote processors.

  While sending proxy patches, only the ones who have gathered radiosity in the current iteration are chosen. Other proxies need not to be sent. The information sent is only the id. of the patch and its gathered radiosity value. In order to get a contiguous memory area, these data are packed by proxy manager. The receiver proxy manager unpacks the message and updates the radiosity values of its local patches according to this information.

  As seen from the Figure 4.10-b, the patch manager triggers the proxy manager to pack and send the proxy patch data, with a message. To go on the next phase a processor waits till getting message from all of the processors to which it has sent proxy patches in step a. This synchronization is a kind of barrier. Processor cannot continue its execution with push & pull phase, since the unreceived messages can include gathered radiosity values of patches that are not contributed to the original patch yet. Synchronization is also required at the end of iteration for load balancing and convergence test.

There we need another synchronization for this operation. A processor cannot process the coming information of proxy patches without finishing evaluation of its all interactions. Because, this may effect the consistency of interactions those yet not processed. These early arrived messages are queued, and processed only after the last interaction is evaluated. Note that, this does not create a problem for the sender processor.

- **step (c): Push & pull phase**

  Parallelization of push & pull operation is relatively easy, since most of the action is performed with local data. The patches having all the subpatches locally present can perform push & pull operations independently. However, some patches may have their subpatches at a remote processor as a result of migration. In order to perform push & pull operation on such patches, we have to communicate these two processors and make them to transfer the required data. This phase is simulated in Figure 4.10-c.

  There are three types of root patches according to the locality of their subpatches:

  - Root patch with completely local subpatches: As explained above, it has no dependence to any other patches, so it can push and pull radiosity directly as in the sequential program.
  - Root patch with some migrated subpatches: Push & pull operation of this type patches depends on their migrated subpatches. First, the root patch starts to push the radiosity. When it encounters a migrated subpatch, it sends a message including the *pushed* radiosity value of the migrated subpatch to its owner processor. The root patch continues to push the radiosity without waiting since it does not require a response to finish the push operation. After finishing the

push operation, it waits responses in a queue till getting all of them. The operations taking place in the other processor are explained in the next item. As soon as a response arrives, relevant subpatch's *pulled* radiosity value is updated. After getting all the responses, the root patch performs pull operation locally, using the *pulled* radiosity values for migrated subpatches.

– Migrated subpatch with remote ancestor patches: These are the subpatches of the patches explained in the previous item. Since their parents are remote, they cannot start push operation independently. They wait messages including their *pushed* radiosity value, from their parent's owner processor. As soon as they receive this message, they perform push & pull operation independently and send the *pulled* radiosity value of root subpatches to their parents.

In order to accelerate this step, first the second type patches start to perform push operations and send requests to their migrated subpatches. These patches will wait responses in a queue. The third type patches perform push and pull operation whenever they receive message from their parents. The least priority belongs to the first type patches since their work is trivial. They perform push & pull operation when processor is idle.

Push & pull operation is free of deadlock, since the requests are not dependent to each other. Each processor deals with different patches. Another subject worth to mention is the synchronization of processors for this operation. The processors are already synchronized in the previous step and therefore it does not cause an important latency.

The speed of this step is relatively fast. All the patches in the environment push and pull the radiosity, giving order $O(N)$ for computational complexity. Communication is required infrequently and produces negligible overhead.

- **step (d): 1. Radiosity and load reduction**

  The radiosity computations of an iteration finish by push & pull phase. Processors prepare themselves for the next iteration if there is. To understand whether the radiosity system is converged or not, all processors broadcast the total radiosity value of their patches. This is simulated in Figure 4.10-d. Change rate of this total radiosity value is a criterion for the convergence test.

Dynamic load balancing requires global communication. Load of each processor should be calculated by estimation and this information must be broadcasted. After all, each processor compares the loads and gives load transfer decision if there is a significant load imbalance among them. This step is explained in Section 4.6.1 in details.

- **step (d) 2. If not converged, balance load**

  If the processors give load transfer decision as a result of significant load imbalance, all of them enter migration step. They construct a load transfer matrix which shows from which processor to which processor and how much load will be transferred. According to this matrix, overloaded processors select their suitable patches and migrate them to the less loaded ones. This operation is explained in Section 4.6.4 in details.

## 4.6 Performance Considerations

Performance of a parallel program greatly depends on the efficiency of its load balancing strategy. In order to produce scalable parallel applications, special care must be taken in the process of load balancing. In our program, we investigated the efficiency of some load balancing strategies and observed their performance for various input data.

Other factors effecting the execution time of a radiosity solution system:

- lower bound on the area of patches (quality of the image),
- maximum polygon number of an octree voxel,
- initial value used for multigridding and its decreasing rate,
- number of rays used for visibility and form factor calculations, and
- size of cache used for the rays mentioned above.

### 4.6.1 Load Balancing

The objective of load balancing algorithms is to distribute the existing work equally to each processor so as to minimize their idle times. Using a good strategy facilitates developing applications scalable in terms of memory and speed. If we want to exploit parallelism, simple or complex, we have to follow a load balancing strategy.

Load balancing strategies can be either static or dynamic type. In static load balancing, we give the decision of load distribution at the beginning of program and do not request a change further. If we can estimate the loads of processors accurately and guarantee the continuity of their balance during execution, we can use this type of strategy. This is mostly true for the programs with static works that have predictable running times and do not introduce new jobs to the system. However, it will be insufficient to consider only initial distribution, when we have dynamically changing data and processes. In such cases, we are forced to revise the load distribution and perhaps give a rearrangement decision of loads, like job transferring. This is dynamic load balancing and is implemented for processes whose loads are changing dynamically. We should consider to transfer some of the tasks of overloaded processors to processors with less loads to balance the load during execution. Initial assignment is still very important, because a good initial assignment can decrease the necessity to dynamic balancing. Although dynamic balancing algorithms are not implemented as easy as static type, and have more overheads such as detecting imbalance, transferring load, they play a great role in producing scalable applications.

Hierarchical radiosity is a very dynamic program where the memory requirements and work load are continuously increasing during its execution. We can observe this dynamic property on a sample run results table Table 4.1 obviously from the changes in every iteration. As the interactions are refined, we see that the number of patches and proxies gets bigger and bigger. Newly introduced interactions cause an increase in the amount of form factor and visibility calculations. By changing some factors listed under the previous title, we may get different timings and results from the Table 4.1 but the same solution. These

Table 4.1: Sample execution results for one of the processors.

| iteration # | total # of patches | total # of proxies | total # of interactions | total time |
|---|---|---|---|---|
| 1 | 137 | 169 | 1214 | 2.4 |
| 2 | 729 | 363 | 2630 | 4.6 |
| 3 | 1235 | 435 | 4340 | 7.0 |
| 4 | 1267 | 617 | 6101 | 11.0 |
| 5 | 1363 | 1125 | 9381 | 17.0 |
| 6 | 1591 | 1825 | 15233 | 25.8 |
| 7 | 1857 | 2081 | 22876 | 39.6 |
| 8 | 2051 | 2235 | 38434 | 62.3 |

are the reasons that make load balancing and therefore parallelization difficult. Therefore we have implemented both static and dynamic load balancing to get a better performance.

### 4.6.2 Load Estimation

To implement a load balancing algorithm, we should estimate the computational load of processors. Since we distribute patches to processors and interactions are associated with patches, we can find load of a processor by summing the computational load caused by all patches on that processor. A trivial approach is to count the number of patches. This method assumes that all patches have uniform and constant computational load. However depending on the interactions that a patch is involved the amount of load might change.

A better approach would be the number of interactions. Load of a patch is estimated as the number of interactions in its interaction list. This method assumes all interactions are equal loaded. This is obviously not correct. For example, an interaction with a light source cannot be considered as equal to an interaction with a dark, small and far patch. Light source interactions have great potential of refinement, in contrast to other interactions. However, this method is efficient for scenes composed of almost equally loaded patches. With considering randomness during distribution of patches, it becomes possible to obtain good results.

To make a better estimation, we should examine all of the possible factors. Load of a patch depends on the following criteria:

- its area,
- its radiance at that moment, and
- its interacting patches'
  - mutual visibility rate,
  - area, and
  - radiance at that moment.

Area is important since the patches are subdivided according to their areas during execution. Subdivision of a triangular patch introduces 4 new interactions that should also be evaluated. Subdividing can be performed to a patch while the newly created subpatches have area bigger than some threshold $A_e$. Maximum number of subpatches that a triangular patch can have is calculated as follows:

```
int maxP(REALTYPE area, REALTYPE A_e) {
int i;
for (i=1; area>A_e; area /= 4, i += 4*i);
return i;
}
```

By the help of this function, upper bound to the number of interactions caused by a patch can be formulated as follows:

$$bound_i = \mathtt{maxP}(\mathtt{area_i}, \mathtt{A_e}) \sum_{j}^{interactions_i} \mathtt{maxP}(\mathtt{area_j}, \mathtt{A_e}) \tag{1}$$

This formulation is true for the cases where all patches are visible and carry at least an amount of radiance more than threshold, leading refinement of all its interactions. However, this is not true for most of the interactions. Only light source interactions can cause such a full refinement. We use another function called `avgP()` by changing the parameters of `maxP()` function to get a reasonable estimation (e.g., by considering a probability factor, we may multiply `i` with a smaller number instead of 4).

Occlusion prevents interactions to be established or be refined at some point. Visibility factor can be used to adjust the calculated bound. Patches with a small visibility rate are not expected to have much interaction. Interactions of fully visible patches have a great potential of refinement. However, no visibility calculation is required after refinement of these interactions since all of them are already expected to be fully visible. In the same way, partially visible interactions with high visibility rate are expected to be refined to create fully visible interactions. As a result, we can say that high and low visibility rates do not contribute too much work to the system.

Subdivision is performed if the radiance carried by the interaction is more than some threshold. After each subdivision, the patches get smaller and become less radiative. This decreases the probability of a further subdivision.

After incorporating these ideas to the estimation bound, our estimation function becomes:

$$est1_i = \mathtt{avgP}(\mathtt{area_i}, \mathtt{A_e}) \sum_{j}^{interactions_i} \mathtt{avgP}(\mathtt{area_j}, \mathtt{A_e}) * \mathtt{F_1}(\mathtt{Vis_{i-j}}) * \mathtt{F_2}(\mathtt{B_i}, \mathtt{B_j}) \tag{2}$$

### 4.6.3 Initial Distribution

Initial distribution, as we have previously mentioned, is very important for both static and dynamic load balancing. In our implementation, to get a better estimation, we performed some computations at the beginning. After building interaction lists, we executed one iteration of radiosity algorithm locally and without refinement to see the properties of patches' interaction lists in a little bit more detail. These properties include the visibility and form factor of interactions and first gathered radiance. Since we neglected refinement part, it does not take too much time. After finishing this step, we choose the input patches with some order and assign them to processors by equalizing their loads.

The order of selection of the input patches is also a factor worth to consider for the efficiency of initial distribution. Three alternatives exist on this subject:

- Randomly: Patches are assigned to processors randomly. Randomness may produce equality, however nearby objects which have bigger interaction, possibly place in different processors increasing the dependency of processors to each other.
- Octree-based: Nearby objects are forced to place in same processors. Since nearby objects have more interaction rather than far ones, the dependencies between processors are expected to be lower. The octree structure built for visibility calculations can be used to detect nearby patches. The criterion of octree-based distribution is only the distance of patches and this may not be a sufficient factor in terms of interaction value.
- In input order: Input order distribution is a kind of spatial distribution as octree-based approach. If the order has a meaning, patches of same objects are expected to be defined successively. Also, different from octree-based distribution, not only near but also interacting objects are highly expected to be defined successively. For example, if we have a scene with 2 rooms, we usually define the objects in one room first, then define the objects in other room. Thus, objects in one room will possibly be assigned to same processor. However, in octree-based distribution, room boundaries are not taken into account. Even there is a wall between them, objects of different rooms may be assigned to same processor just because they are near in octree-based approach.

  Note that, if the order does not have a meaning, input order distribution may result worse than random distribution, since we are restricted to select always the next patch in the order when distributing them.

Random and input order distribution greatly depend on how we gave order

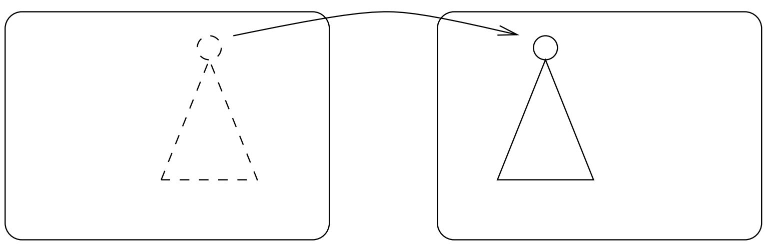


Figure 4.11: Patch migration.

to input patches. That is, by changing the order of input patches, we can get different performance results for the same scene. This may give us the best performance as well as the worst one.

There is one thing worth to say about the effect of interacting patches' distance on the load of patches. Evaluating interactions of far objects must not be expected to be cheaper than evaluating interactions of near objects, in terms of time. Although far objects have small form factors, calculating their visibility factor takes more time compared to the interactions of nearer objects. Because, there will be more candidates of intervening objects and more octree cells to check for far objects. This is an important feature that increases the efficiency of the random selection method.

Comparisons of these methods are presented in the next chapter together with performance results.

### 4.6.4 Patch Migration

Strictly speaking, migration is the process of moving a patch from one processor to another together with its required data. Figure 4.11 and Figure 4.12 illustrate this operation. The aim of migrating a patch is to balance the load of processors by transferring jobs originating from the patch. It is an inevitable operation when dynamic load balancing is considered. If there is significant load imbalance between processors, then the overloaded processors must get rid of them by transferring to the idle ones.

Handling migration is not simple. Some questions arise here which must be

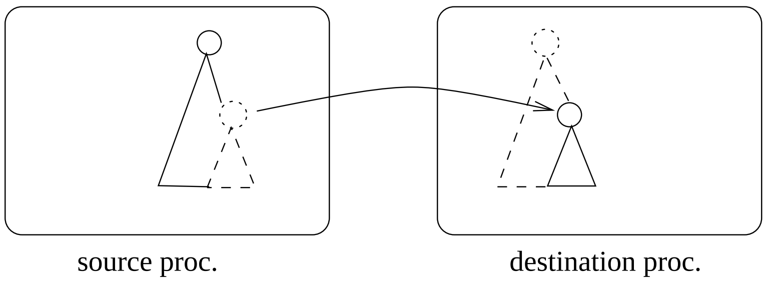


Figure 4.12: Subpatch migration.

carefully considered during the design of the migration operation:

- When to migrate? Decision of migration should be given only if there is a significant load imbalance between processors. Since the load calculation is based on estimations, an error rate should be considered to decide the level of imbalance. Otherwise, we might be transferring data through the wrong direction!

- Which to migrate? How many to migrate? We should estimate load of each patch and choose the appropriate one(s) to send in order to minimize the overhead of migration. Subpatch migration (Figure 4.12) must also be allowed since we may encounter difficulties during selecting the patches to be migrated, especially when the number of processors is relatively more than the number of input patches.

- How to migrate? Migration is not performed simply by directly transferring patch and interaction list information. As we have explained before, in order to ease and speed up accesses, processor dependent local id.s are used for patches. Therefore, in order to migrate a patch with its interaction list information correctly, we should provide id.s of both patches and interactions local to the processor to be migrated. Also migration of subpatches creates dependencies between sender and receiver processors, which can cause delays on some radiosity computations.

- Where to migrate? Overloaded processors must transfer their load to less loaded processors but a heuristic way must be developed to give decision of which processors should send load to which processors. In order to decrease the overhead caused by migration, this decision is expected to minimize the dependencies between processors.

In our design, migration is performed within a single step separate from radiosity computations. Migration operation starts with detecting significant load imbalance between processors. Each processor knows the loads of other processors and decides whether if it sends load, receives load or does nothing. The processors that will transfer their loads, select their the most appropriate patch(es) and send them to less loaded ones. Besides patch migration, subpatch migration is also allowed so as not to restrict patch selection algorithm. But patch migration is preferred due to the reasons explained later.

After choosing the suitable patches, they are migrated to their new owner processor without their interaction lists. The interaction lists cannot be migrated directly since they carry localized information. The processor directs these interactions to the processor that the chosen patch is migrated to and interaction lists are reconstructed in the new owner processor. In order to inform other processors about this migration operation, messages are broadcasted by the new owner processor. All processors start next iteration with normal execution, as if there was no migration. However, subpatch migration may lead some problems which are explained later.

Let's see how a migration is performed in details:

Migration decision is given by load balance algorithm which is executed at the beginning of wanted iteration of radiosity solution process. The output of the load balance detection algorithm is a matrix showing from which processor to which processor and how many load should be transferred. This is a global information since all the processors have the same input to the load balance program. But after getting this matrix, the processors return to their private tasks. According to this matrix, a processor can be any of these types: either a source processor for migration, a destination processor for migration or none. The source processors select appropriate patch or patches to send to destination processors and inform the other processors about the migration operation. The destination processors receive the selected patches and locate them into new memory areas.

In order to perform load transmission, the source processor selects appropriate patch or patches according to their loads and the load supposed to be

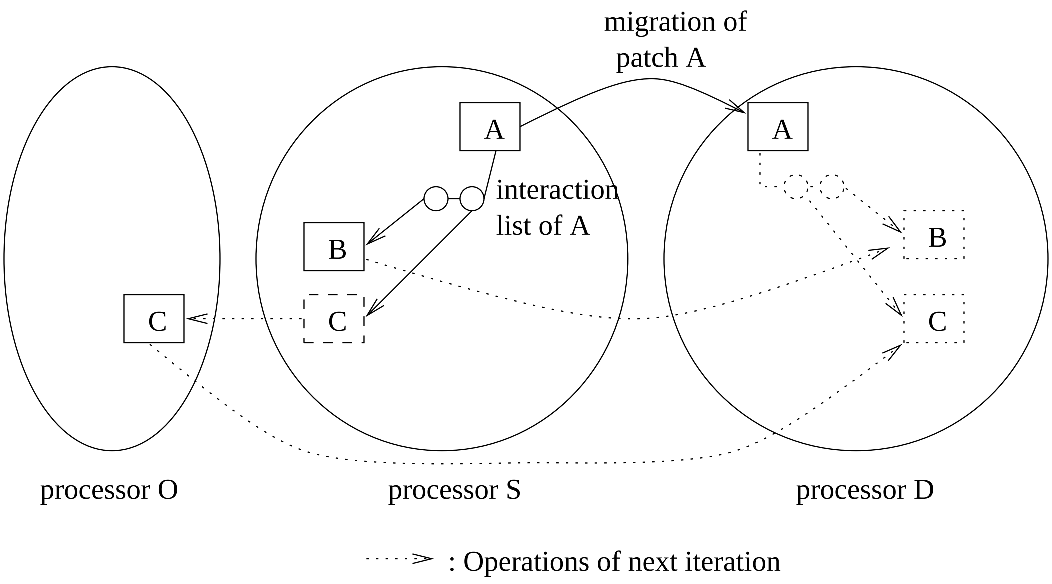


Figure 4.13: Moving interactions of a migrated patch.

transferred. The patch whose load is the nearest one to the required load is selected each time till the remaining required load becomes 0 or a negligible value. It is preferred to minimize the number of patches to migrate by selecting the most appropriate ones. However it is mostly the case that more than one patch are selected to migrate from any processor to any processor.

After selecting the patches to be migrated, the source processor packs and sends their data. The destination processors take these data, unpack them and return local id.s for all patches to the source processor immediately. This information will help the source processor to direct the interactions related with the migrated patches easily.

Note that the source processor does not send interaction lists of migrated patches in this step. Migration is not a simple operation. Because the information carried on a patch to be migrated is required to be validated by the processor that it will migrate to. Since the interactions are localized by proxies, transferring them will require special care.

Interactions are handled at source processors after receiving all proxy patches which are candidates of migrated patches' interacting patches, from every other processors. At this point, patch managers of the source processors check all the interactions of migrated patches and direct them to the new processor by revising some interaction information. Let's explain it with an example illustrated in Figure 4.13. Patch $A$ will migrate from processor $S$ to processor $D$, and patch $B$ and proxy patch $C$ is in the interaction list of patch $A$. Patch manager of processor $S$ tells patch $B$ and proxy $C$ to go to the processor $D$ in the next iteration, find patch $A$ there and add themselves to its interaction list. The next iteration, patch $B$ and proxy $C$ will go to the processor $D$ from their owner processors, find patch $A$ immediately with its local id., and add themselves to its interaction list. So that these two interactions will be evaluated in that iteration without any loss. Then the program will continue its execution in the same way as if the patch is not migrated.

Impacts of a migration operation on source, destination and other processors are obviously different. Here are the roles of these processors during a migration operation:

- **Step (1): Common step**
  - Each processor has an array including loads of all processors. They construct the same load transfer matrix by executing the same decision mechanism with same inputs. According to this matrix, processors are either source, destination processors for migration, or none of them.
- **Step (2): For source processors**
  - Find suitable patch or patches to transfer according to the load transfer matrix.
  - Send all of the selected patches' and their subpatches' data to the destination processors. These data do not include their interaction lists.
  - Receive local id.s from destination processors of the migrated patches. These information will help us to direct the interactions of the migrated patches.
  - Send migrated patches to other processors to inform them about the migration operation.

- Send required patches to other source processors, so that the source processors will direct interactions of their migrated patches.
- Receive proxies from the rest of the system.
- Direct interactions of migrated patches by deleting the old one and creating a new interaction with new information. The newly created interactions are kept in migrated interactions list.

- **Step (2): For destination processors**
  - Receive patches from source processors which are determined by load transfer matrix. For the first time visitors allocate new memory, for other patches convert their old proxy information into local patch type.
  - Send new local id.s information to the source processor of migrated patches.
  - Send all required patches to source processors, so that the source processors will direct interactions of their migrated patches.

- **Step (2): For other processors**
  - Send all required patches to source processors, so that the source processors will direct interactions of their migrated patches.

- **Step (3): Common step**
  - Receive patches from all source processors, update migration information.
  - Send all changed proxies to their owner processors.
  - In the next iteration, the migrated interaction lists will be included in packed patch data, so that the interaction lists of recent migrated patches will be reconstructed without loss.

### Discussion

Migration step is a communication step which does not include any radiosity computations. The source processors do the major part of migration, by sending migrated patches to destination processors and dealing with their interaction lists.

The main reason for separating this operation from radiosity computations is maintaining highly localized data. All patches and their interaction lists keep pointers to memory areas local to the processor that they are belong to. It is clear that using these pointers in the processor where the patch will migrate to, does not make any sense. Although this makes migration difficult, we get its profit in other steps very much.

As an alternative to the separate step migration method, we can also perform migration operations concurrent with an ordinary iteration of radiosity computations. In such a case, it would not be easy to handle transferring interaction lists. Remote patches, in the interaction list of migrated patch will come to the source processor as proxy patches. After all these patches place in the source processor, it becomes possible to send them to the destination processor as interactions of migrated patch. In order to solve the local index problem, all proxies bring also their index local to the destination processor, at the beginning of the step. There is no a big problem so far. If the local patches or proxy patches have valid local id.s for destination processor (i.e., they have visited the destination processor at least once before), local indexing does not create problem. However, for the first time visitors, new memory locations are allocated in the destination processor and this may threat consistency of these patches. Same patch can be transferred to the destination processor from different processors. In each transfer, destination processor allocates different memory locations for the same proxy patch resulting inconsistency for it. To overcome this problem, either we can direct these interactions to destination processor as our current method which may lead delayings for that interaction, or find out if the current patch is visited the destination processor at the same iteration by searching the newly created patches array. Directing interactions

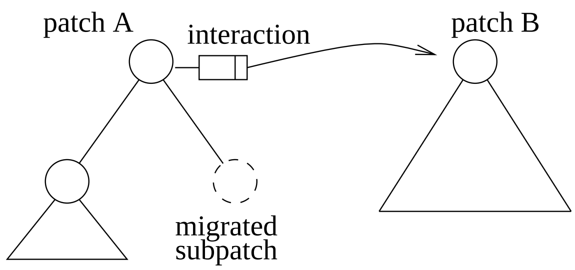


Figure 4.14: Subpatch migration problem.

cause postponing evaluation of them to next iteration. Since we perform radiosity computations in the current iteration, delaying some interactions would cause late convergence. Searching array whenever we encounter a first visitor patch is the only way to overcome this invalid local id. problem. Programming this method tends to be very complicated and may produce erroneous results if we take the obligation of transferring patch with their subtree up to a level into account.

Due to performance considerations, we also have to restrict selection of appropriate patches to migrate. If a subpatch is migrated once, its subpatches are not permitted to migrate in further iterations. Such an operation may cause long waits especially in the push & pull phase which may require a varying size of ring communication.

In order to increase the alternatives when selecting appropriate patches to migrate, source processors are permitted to choose subpatches. This is very important especially for the cases where there are less input patches with respect to processors. Therefore, processors can migrate subpatches up to a level, without deforming hierarchy of its tree.

Subpatch migration creates dependencies between source and destination processors during radiosity computations. This dependency is both in the gathering radiosity operation and in the push & pull phase. After subpatch migration, parent and child become located at different processors. While evaluating interactions of parent patch, the interaction may subject to be refined. Refinement of an interaction causes passing the interaction to the children of the owner patch. If one of the children does not exist because of migration, we fail to evaluate the interaction in the current processor. This case is presented in the Figure 4.14. Patch $A$ owns the interaction, and refinement of this interaction causes the migrated subpatch to own the refined interaction. Migrated subpatch is no more a local patch and cannot behave like a local patch. It is located at another processor and may not have up-to-date information in the current processor. The interaction between migrated subpatch and patch $B$ causes problem at this point. To overcome this problem:

- Migrated subpatches always send proxies to their original processors, and keep their data up-to-date.
- Proxies are allowed to own and evaluate interactions.

# Chapter 5

# Performance Evaluation

This chapter presents results of performance studies conducted to understand the impact of various design and implementation decisions on the performance. These decisions include load estimation methods, initial patch distribution, subdivision depth limit and patch migration. The results are obtained by several runs for various input scenes on the Parsytec distributed memory machine.

Unless stated otherwise, all time values are in seconds and all size values are in bytes.

## 5.1 The Input Scenes

Hierarchical radiosity algorithms cannot use models which have been prepared for other radiosity algorithms. Because, hierarchical radiosity algorithms work with models which consist of undivided surface patches. However, for example progressive radiosity algorithms require all input patches to be divided in advance into fine resolutions. Their models consist of too many subdivided patches, and are useless for hierarchical radiosity. In hierarchical approach, the input patches are accepted as undivided and divided during radiosity calculations dynamically.

We used 4 different input scenes during our performance study which have been prepared according to hierarchical approach. Their characteristics are listed below:

- Scene 1: (see Figure 5.1 and Figure 5.5) The scene has 30 input patches. There is a box which has a rectangular trapezoid prism and a triangular prism inside. Since the patches are relatively small sized and less in amount, we have not much computation to account for parallelization.
- Scene 2: (see Figure 5.2) The scene has 216 input patches. It represents an office with some furniture. An important characteristic of the scene is that, visibility rates of patches are relatively high. This increases the amount of interactions between patches. Another important feature is its irregularity in terms of surface sizes. Small numbered large patches create problems during initial distribution since they have a great potential of interaction with the rest of the objects in the scene.
- Scene 3: (see Figure 5.3) There are 174 input patches in this scene. A house plan with 8 rooms is represented. Although the scene has less input patches, it has more detail and requires more effort to be rendered than Scene 2. Not most of the patches can see each other due to walls. Since the amount of big patches is relatively much than available processors, its initial distribution is easier than Scene 2.
- Scene 4: (see Figure 5.4 and Figure 5.6) There are 252 input patches in this scene. It has same plan with Scene 3 with some extra furnitures. Due to these newly added objects, it requires the longest time to be rendered.

In Table 5.1, some statistical information about scenes are presented. Values are obtained by running the program sequentially. Note that parallel running changes only the time values. As seen from the table, number of final patches are very big with respect to the number of input patches. This observation shows the attractiveness of hierarchical method beside the other methods. Also, number of interactions are very low if it is compared with the square of final patches count which is equal to the number of potential interactions. Convergence rate is the change percentage of patch radiosities in the current iteration.

Table 5.1: Scenes used in performance studies (results are for one processor).

| Scenes | input patches | final patches | total interaction | convergence rate | iteration | preproc. time | comput. time |
|---|---|---|---|---|---|---|---|
| Scene 1 | 30 | 4677 | 90891 | 0.007 | 10 | 0.5s | 57.8s |
| Scene 2 | 216 | 6610 | 187818 | 0.016 | 9 | 9.5s | 171.8s |
| Scene 3 | 174 | 13608 | 348508 | 0.006 | 9 | 4.8s | 198.0s |
| Scene 4 | 252 | 14170 | 350622 | 0.006 | 9 | 13.5s | 213.3s |

## 5.2 Impact of Load Estimation Methods

The Table 5.2 and Table 5.3 show timings for various load estimation methods discussed in Section 5.2. For each load estimation method, we ran the program on different number of processors for each input scene and measure the execution time. Table 5.2 shows the timings where the patches are distributed by the input order method. As discussed in the Section 5.2 number of patches as an estimation for processor load is observed to be inferior than others. Particularly for Scene 2, distributing equal number of patches to processors performs very poorly because probably large patches were assigned to the same processor. However, number of interactions method or the load formula method do not threat patches equally and more intelligent decisions can be done based on the work to be done. The results show that the latter two methods perform much better than the naive number of patches method and our load estimation formula seems to be superior among all. Table 5.3 shows the timings of the same experiment for the octree-based distribution. Again the results are similar to the previous experiment.

## 5.3 Impact of Initial Patch Distribution

In this part, we compared three different patch distribution methods which are explained in Section 4.6.3. They are namely random, input order and octree-based distribution. Table 5.4 shows timings and Table 5.5 shows communication volumes of sample runs.

Table 5.2: Comparison of load estimation methods (input order patch distributing method) (p: according to patch number, i: according to interaction number, f: according to the presented formula).

| Scenes | Seq. time | # P | p | i | f |
|---|---|---|---|---|---|
| Scene 1 | 57.8 | 2 | 58.2 | 34.9 | 35.2 |
| | | 4 | 35.3 | 20.0 | 17.5 |
| Scene 2 | 171.8 | 2 | 177.3 | 172.9 | 141.9 |
| | | 4 | 173.6 | 158.4 | 80.8 |
| | | 8 | 177.3 | 94.0 | 61.2 |
| | | 16 | 176.7 | 61.0 | 60.7 |
| Scene 3 | 198.0 | 2 | 129.0 | 119.0 | 117.3 |
| | | 4 | 90.5 | 67.8 | 60.0 |
| | | 8 | 60.6 | 50.0 | 39.6 |
| | | 16 | 46.4 | 31.2 | 34.0 |

Table 5.3: Comparison of load estimation methods (octree-based patch distribution) (p: according to patch number, i: according to interaction number, f: according to the presented formula).

| Scenes | Seq. time | # P | p | i | f |
|---|---|---|---|---|---|
| Scene 1 | 57.8 | 2 | 40.6 | 35.0 | 34.8 |
| | | 4 | 25.8 | 21.3 | 21.3 |
| Scene 2 | 171.8 | 2 | 96.6 | 94.9 | 113.4 |
| | | 4 | 63.6 | 61.8 | 77.4 |
| | | 8 | 60.4 | 60.2 | 60.4 |
| | | 16 | 60.7 | 61.6 | 61.2 |
| Scene 3 | 198.0 | 2 | 114.9 | 121.4 | 116.7 |
| | | 4 | 69.2 | 64.9 | 60.7 |
| | | 8 | 65.7 | 44.1 | 40.5 |
| | | 16 | 49.0 | 34.8 | 29.2 |

Table 5.4: Timings for sample runs of different patch distribution methods.

| Scenes | Seq. time | # P | random | input order | octree-based |
|---|---|---|---|---|---|
| Scene 1 | 57.8 | 2 | 34.2 | 34.8 | 34.2 |
| | | 4 | 18.7 | 17.5 | 21.0 |
| | | 8 | 15.9 | 15.8 | 16.0 |
| | | 16 | 17.3 | 17.5 | 17.7 |
| | | 24 | 18.0 | 18.8 | 18.8 |
| Scene 2 | 171.8 | 2 | 125.8 | 139.7 | 110.8 |
| | | 4 | 62.8 | 79.4 | 76.6 |
| | | 8 | 59.8 | 60.7 | 60.0 |
| | | 16 | 60.6 | 60.2 | 60.0 |
| | | 24 | 61.6 | 61.7 | 62.0 |
| Scene 3 | 198.0 | 2 | 106.1 | 114.5 | 116.7 |
| | | 4 | 61.3 | 59.5 | 60.7 |
| | | 8 | 41.7 | 39.1 | 40.5 |
| | | 16 | 32.9 | 34.0 | 29.2 |
| | | 24 | 25.3 | 25.8 | 25.5 |
| Scene 4 | 213.3 | 2 | 117.4 | 134.3 | 126.8 |
| | | 4 | 70.6 | 66.8 | 65.0 |
| | | 8 | 48.2 | 41.5 | 39.5 |
| | | 16 | 31.0 | 30.0 | 28.9 |
| | | 24 | 30.9 | 31.5 | 28.6 |

If we look at the speed up of any distribution method we observe that up to four processors, reasonable speed ups were obtained. After four processors, there is no significant improvement in execution time for Scene 1 and Scene 2 is observed. This is because there are not enough number of patches at the beginning of the execution to load processors. However, as patches are divided into smaller patches, the load can be balanced at later steps by dynamic load mechanism which will be discussed in the next section.

We observe also that none of the initial distribution methods is superior to any other one. This result is due to highly irregular structure of the computations. A static distribution based on initial information of the patches does not guarantee load balance throughout the execution.

Random distribution is slightly worse than others. Because it creates more communication as shown in Table 5.5.

Table 5.5: Communication volumes for sample runs for different patch selection methods.

| Scenes | # P | random | input order | octree-based |
|---|---|---|---|---|
| Scene 1 | 2 | 495K | 503K | 408K |
| | 4 | 1013K | 984K | 963K |
| | 8 | 1338K | 1268K | 1357K |
| | 16 | 1537K | 1380K | 1451K |
| | 24 | 1544K | 1433K | 1494K |
| Scene 2 | 2 | 657K | 572K | 654K |
| | 4 | 1537K | 1367K | 1372K |
| | 8 | 2349K | 2300K | 2304K |
| | 16 | 3085K | 2865K | 2857K |
| | 24 | 3407K | 3059K | 3046K |
| Scene 3 | 2 | 1367K | 415K | 470K |
| | 4 | 2282K | 943K | 1431K |
| | 8 | 3533K | 2662K | 2549K |
| | 16 | 5003K | 4300K | 4231K |
| | 24 | 5644K | 5132K | 5152K |
| Scene 4 | 2 | 1399K | 218K | 518K |
| | 4 | 2728K | 1165K | 1620K |
| | 8 | 3924K | 2550K | 3035K |
| | 16 | 5677K | 4783K | 4883K |
| | 24 | 6517K | 5947K | 6041K |

## 5.4 Impact of Dynamic Load Balancing

We conducted experiments with dynamic load balancing. In these experiments after a number of iterations the program enters into migration step to rebalance the load of the processors by moving patches from overloaded processors (as explained in Section 4.6.4).

Table 5.6, Table 5.7 and Table 5.8 compares the performance results for initial distribution and dynamic load balancing schemes. Especially from the timings of Scene 1 and Scene 2, we observe that static load balancing does not produce scalable results as the processor number increases. However with dynamic load balancing, consistent improvements in execution time is observed as the number of processors increases. For example as seen from Table 5.6 for Scene 4 10.6 speed up is obtained in 24 processors. The results show the necessity and the efficiency of patch migration operation.

## 5.5 Impact of Patch Subdivision Depth Limit

One of the design decisions of our parallel algorithm was to put a limit on patch subdivision (see Section 4.5.2).

Table 5.9 shows timings for different level subdivision limit 1 to 4. If the limit is $i$ we let the patches to subdivide into at most $i$ level subdivisions within a single iteration. Fourth column is the ratio of the last iteration's total radiosity to the previous iteration's total radiosity. It is a kind of convergence speed.

According to results summarized in the table, selecting a high limit does not produce better results, instead increases communication volume. Note that the subdivision depth limit is also the depth limit of proxy transfer. Therefore, in high depth limits, proxy patches are transferred with bigger subtree for nothing. Also selecting limit as 1 is very restrictive and decreases the convergence speed. Therefore we selected the limit as 2 for the performance studies presented in this chapter.

Table 5.6: Statistics for runs including migration with random patch distribution( *: no migration required, **: migration failed within given limits).

| Scenes | Seq. time | P # | time without migration | time with migration | comm. volume | max. migration level |
|---|---|---|---|---|---|---|
| Scene 1 | 57.8 | 2 | 34.2 | 31.9 | 469K | 2 |
| | | 4 | 18.7 | 18.1 | 1127K | 3 |
| | | 8 | 15.9 | 11.3 | 2029K | 3 |
| | | 16 | 17.3 | 8.9 | 3435K | 4 |
| | | 24 | 18.0 | **18.0 | | |
| Scene 2 | 171.8 | 2 | 125.8 | 96.6 | 737K | 1 |
| | | 4 | 62.8 | 61.6 | 1608K | 1 |
| | | 8 | 59.8 | 35.3 | 3131K | 3 |
| | | 16 | 60.6 | 22.9 | 4960K | 3 |
| | | 24 | 61.6 | 18.9 | 6356K | 4 |
| Scene 3 | 198.0 | 2 | 106.1 | *106.1 | | |
| | | 4 | 61.3 | *61.3 | | |
| | | 8 | 41.7 | 40.1 | 3912K | 2 |
| | | 16 | 32.9 | 23.5 | 5911K | 3 |
| | | 24 | 25.3 | 19.4 | 7396K | 3 |
| Scene 4 | 213.3 | 2 | 117.4 | 116.4 | 1399K | 1 |
| | | 4 | 70.6 | 68.4 | 2796K | 1 |
| | | 8 | 48.2 | 42.9 | 4234K | 1 |
| | | 16 | 31.0 | 27.0 | 6251K | 1 |
| | | 24 | 30.9 | 20.3 | 8504K | 3 |

Table 5.7: Statistics for runs including migration with input order patch distribution (*: no migration required.)

| Scenes | Seq. time | P # | time without migration | time with migration | comm. volume | max. migration level |
|---|---|---|---|---|---|---|
| Scene 1 | 57.8 | 2 | 34.8 | 31.3 | 551K | 2 |
| | | 4 | 17.5 | 17.4 | 1033K | 3 |
| | | 8 | 15.8 | 11.4 | 1954K | 3 |
| | | 16 | 17.5 | 8.8 | 3177K | 3 |
| | | 24 | 18.8 | 8.9 | 4529K | 4 |
| Scene 2 | 171.8 | 2 | 139.7 | 113.8 | 694K | 1 |
| | | 4 | 79.4 | 59.8 | 1584K | 1 |
| | | 8 | 60.7 | 34.7 | 2910K | 3 |
| | | 16 | 60.2 | 24.0 | 4783K | 3 |
| | | 24 | 61.7 | 18.8 | 6056K | 4 |
| Scene 3 | 198.0 | 2 | 114.5 | *114.5 | | |
| | | 4 | 59.5 | *59.5 | | |
| | | 8 | 39.1 | 38.4 | 2834K | 1 |
| | | 16 | 34.0 | 22.5 | 4982K | 3 |
| | | 24 | 25.8 | 19.6 | 7077K | 3 |
| Scene 4 | 213.3 | 2 | 134.3 | *134.3 | | |
| | | 4 | 66.8 | *66.8 | | |
| | | 8 | 41.5 | *41.5 | | |
| | | 16 | 30.0 | 24.7 | 5801K | 2 |
| | | 24 | 31.5 | 21.8 | 8100K | 2 |

Table 5.8: Statistics for runs including migration with octree-based patch distribution (*: no migration required).

| Scenes | Seq. time | P # | time without migration | time with migration | comm. volume | max. migration level |
|---|---|---|---|---|---|---|
| Scene 1 | 57.8 | 2 | 34.2 | 33.3 | 535K | 1 |
| | | 4 | 21.0 | 17.8 | 1120K | 2 |
| | | 8 | 16.0 | 11.9 | 1909K | 4 |
| | | 16 | 17.7 | 9.2 | 3343K | 4 |
| | | 24 | 18.8 | 9.3 | 4647K | 4 |
| Scene 2 | 171.8 | 2 | 110.8 | 96.9 | 672K | 1 |
| | | 4 | 76.6 | 55.0 | 1622K | 2 |
| | | 8 | 60.0 | 35.5 | 2876K | 3 |
| | | 16 | 60.0 | 24.1 | 4988K | 4 |
| | | 24 | 62.0 | 18.9 | 6380K | 4 |
| Scene 3 | 198.0 | 2 | 116.7 | *116.7 | | |
| | | 4 | 60.7 | *60.7 | | |
| | | 8 | 40.5 | 35.5 | 2972K | 3 |
| | | 16 | 29.2 | 23.3 | 5218K | 3 |
| | | 24 | 25.5 | 19.0 | 6943K | 3 |
| Scene 4 | 213.3 | 2 | 126.8 | *126.8 | | |
| | | 4 | 65.0 | *65.0 | | |
| | | 8 | 39.5 | 35.5 | 3292K | 2 |
| | | 16 | 28.9 | 26.1 | 5308K | 2 |
| | | 24 | 28.6 | 22.1 | 8442K | 4 |

Table 5.9: Statistics for sample runs with different subdivision depth limits (for two processors).

| Scenes | depth limit | time | rad. change ratio | # of interactions | comm. volume |
|---|---|---|---|---|---|
| Scene 1 | 1 | 32.0 | 0.0070 | 86405 | 264K |
| | 2 | 35.3 | 0.0076 | 90891 | 467K |
| | 3 | 35.5 | 0.0077 | 92409 | 507K |
| | 4 | 35.6 | 0.0077 | 92430 | 538K |
| Scene 2 | 1 | 106.3 | 0.0151 | 175032 | 415K |
| | 2 | 112.3 | 0.0162 | 187818 | 654K |
| | 3 | 114.5 | 0.0169 | 192569 | 747K |
| | 4 | 116.8 | 0.0169 | 192584 | 785K |
| Scene 3 | 1 | 108.2 | 0.0065 | 341259 | 327K |
| | 2 | 117.3 | 0.0065 | 348508 | 470K |
| | 3 | 119.2 | 0.0065 | 349338 | 682K |
| | 4 | 119.2 | 0.0065 | 349341 | 1070K |

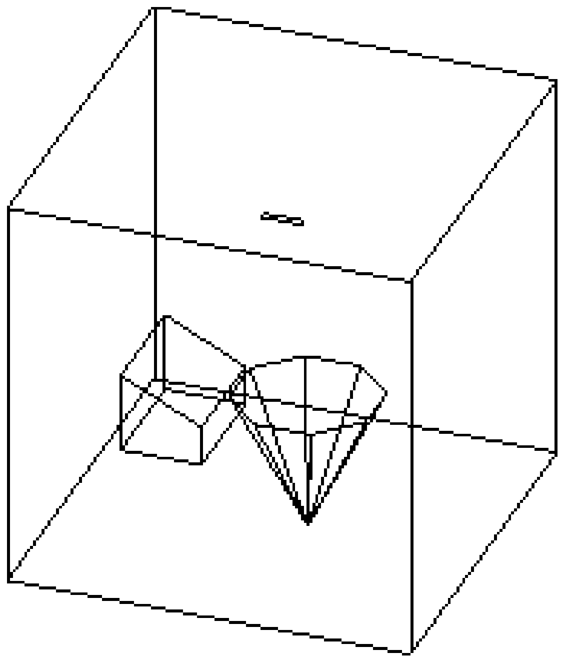

Figure 5.1: Scene 1, wireframe picture.

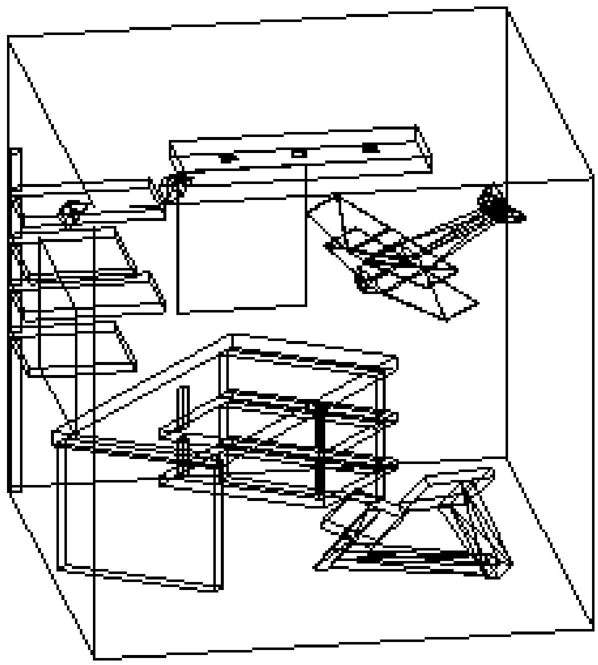

Figure 5.2: Scene 2, wireframe picture.

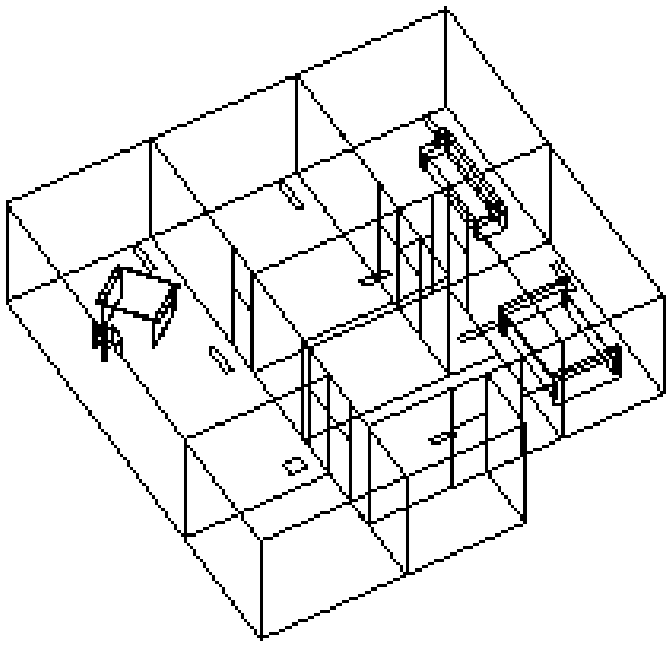

Figure 5.3: Scene 3, wireframe picture.

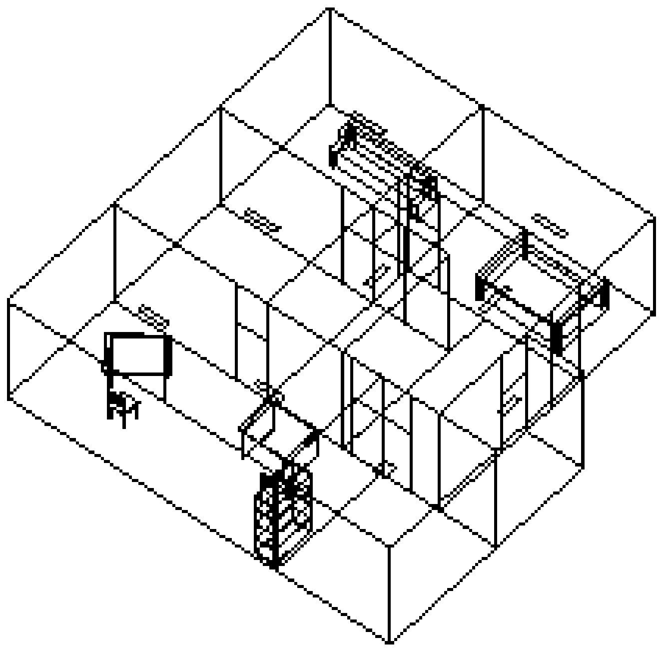

Figure 5.4: Scene 4, wireframe picture.

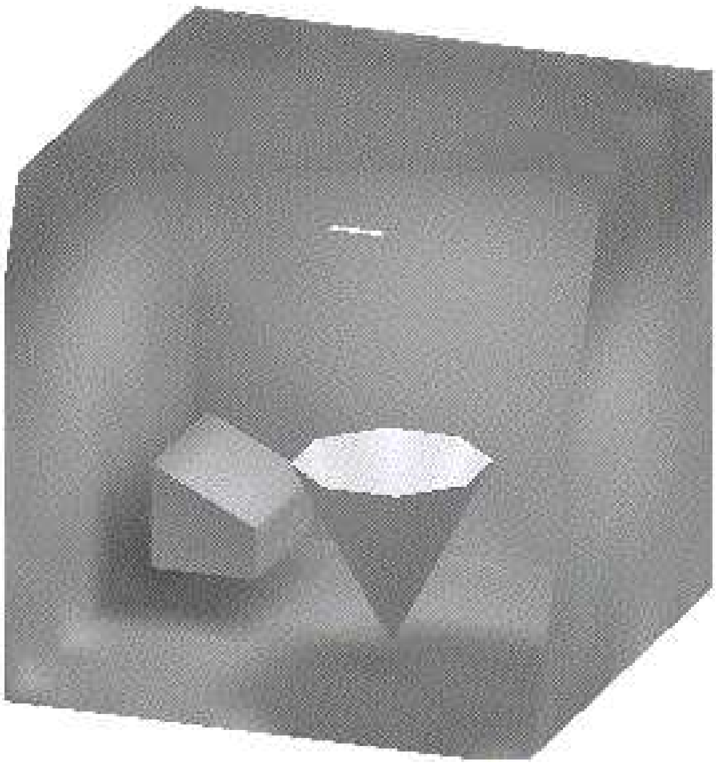

Figure 5.5: Scene 1, shaded image.

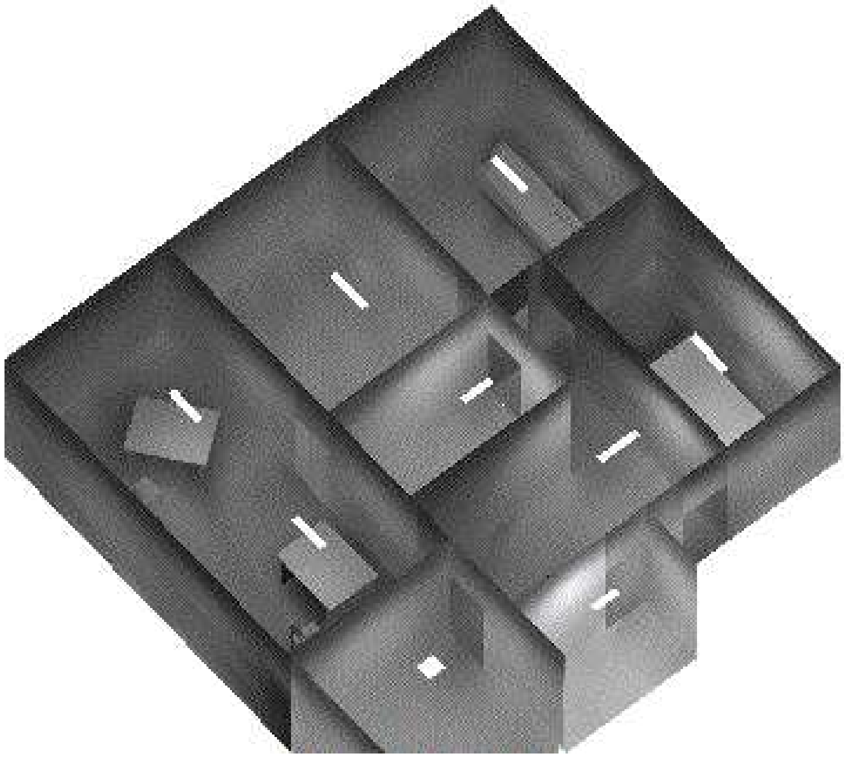

Figure 5.6: Scene 4, shaded image.

# Chapter 6

# Conclusions and Future Work

In this thesis, we have investigated parallelism for hierarchical radiosity algorithms on distributed memory computers. We have designed and implemented a parallel hierarchical radiosity algorithm using message driven library of Charm++. In order to make Charm++ available on our parallel machine Parsytec, we ported it by rewriting its machine interface modules with EPX, Parsytec's native message-passing library. Then we have conducted a performance study to measure the efficiency of our parallelization scheme. This is the summary of our work presented in this thesis.

It is always difficult to develop parallel applications. Special care must be taken when designing issues related with parallelization such as initial load distribution, synchronization, scheduling, load balancing, data transfer policy. Sometimes it may be very difficult for the application to be scalable even with a good design, if the data manipulated is changing dynamically. This property causes load imbalances between processors during execution and processors are forced to balance their load dynamically. Unfortunately this is the case for hierarchical radiosity approach. Different from most of the parallel applications, hierarchical radiosity deals with dynamically growing data and this makes it difficult to parallelize. This was the major problem that we encountered during our parallelization study. Some of the other problems are stated in Section 4.2.

In order to give a fair decision for the initial distribution of patches, we have developed a mechanism to estimate load of each patch. Estimation based on only interaction numbers and estimation based on formula were observed to be efficient. Patches are distributed to processors in different ways, either randomly or octree-based or in input order. Although none of them is observed to be superior to others, differences of their performance results are worth to consider.

Using proxy patches as representatives of remote patches is an important decision of our design. The advantages gained by this idea are: a) Communication volume is decreased since a proxy is sent only once for all its interacting patches. b) Consistency of remote patches is preserved since there is at most one proxy for a patch at each processor. c) It is provided that the radiosity functions operate as if all the patches are local. d) And requesting step for remote patches is removed since they know which processor needs its data. We have assigned managers responsible for proxy patches and local patches to ease their manipulation.

As seen from the static load balancing results, we were forced to perform a dynamic load balancing strategy. Due to its importance for parallelism, we have spent greatest effort on this part. The strategy we followed for dynamic load balancing is migrating the patches and their interactions from overloaded processors to less loaded ones. This operation is performed when a significant load imbalance between processors is detected. After detecting imbalance, all of the processors enter load balancing phase instead of ordinary iterations. After migrating patches and their interactions, processors return their normal execution with their changed data. Necessity and efficiency of patch migration operation can easily be observed from performance results.

For the radiosity to be usable in commercial applications, still too much work needs to be done besides parallelization. There are several areas which are open to further research.

- Clustering is a must for radiosity algorithms. It is an extension of hierarchical approach and more close to N-body algorithms.

- Hierarchical radiosity can be investigated to integrate with progressive radiosity method to exploit its advantages especially for parallel processing.
- For a better image quality, shadow boundaries must be handled separately as in discontinuity meshes. Also, both of the diffuse reflector and mirror-like surfaces must be handled together.

# Bibliography


[ACO96] C. Aykanat, T. Capin, and B. Ozguc. A parallel progressive radiosity algorithm based on patch data circulation. *Computers & Graphics*, 20(2):307–324, 1996.

[ACSG98] M. Atun, I. Cengiz, R. Sireli, and A. Gursoy. Implementation of Converse interoperable programming environment on Parsytec CC multicomputers. In *Advances in Computer and Information Sciences'98*, IOS/Ohmsa, pages 367–374, Oct 1998.

[CAO93] T. Capin, C. Aykanat, and B. Ozguc. Progressive refinement radiosity on ring-connected multicomputers. In *Proceedings of IEEE 1993 Parallel Rendering Symposium*, ACM SIGGRAPH, pages 71–76, 1993.

[CCWG88] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, Aug 1988.

[CHRM97] *Charm++ Programming Manual*, 1997. Department of CS, University of Illinos at Urbana-Campaign.

[CON96] *Converse Programming Manual*, 1996. Department of CS, University of Illinos at Urbana-Campaign.

[CW93] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, NY, 1993.

[DBSW97] Ph. Dutre, Ph. Bekaert, F. Suykens, and Y. D. Willems. Bidirectional radiosity. In *8th Eurographics Workshop on Rendering*, 1997.

[ECGS92] T. V. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of 19th Ann. Int'l Symp. Computer Architectures*, pages 256–266, 1992.

[EPX95] *Embedded Parix, Software Documentation*, 1995. Parsytec GbmH.

[FY97] C-C. Feng and S-N. Yang. A parallel hierarchical radiosity algorithm for complex scenes. In *IEEE Parallel Rendering Symposium*, pages 71–77, 1997.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, R. Manchek W. Jiang, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* OAK Ridge National Laboratory, Knoxville, TN, 1994.

[GBM94] R. Garmann, C-A. Bohn, and H. Muller. Parallel hierarchical radiosity on the CM-5. Technical Report 557, Department of Computer Science, Dortmund University, 1994.

[GRS95] P. Guitton, J. Roman, and G. Subrenat. Implementation results and analysis of a parallel progressive radiosity. In *IEEE Parallel Rendering Symposium*, pages 31–38, 1995.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, Jul 1984.

[HSA91] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, Jul 1991.

[HSD94] N. Holzschuch, F. Sillion, and G. Drettakis. An efficient progressive refinement strategy for hierarchical radiosity. In *Fifth Eurographics Workshop on Rendering*, ACM SIGGRAPH, Darmstadt, Germany, Jun 1994.

[KAO97] T. M. Kurc, C. Aykanat, and B. Ozguc. A parallel scaled conjugate-gradient algorithm for the solution phase of gathering radiosity on hypercubes. *The Visual Computer*, 13:1–19, 1997.

[KBJK96] L. V. Kale, M. Bhandarkar, N. Jagathesan, and S. Krishnan. Converse: An interoperable framework for parallel programming. In *Proceedings of IPPS'96*, pages 212–217, 1996.

[KG95] L. V. Kale and A. Gursoy. Reuse and efficiency with message-driven libraries. In *Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 738–743, 1995.

[KK93] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, ACM Sigplan Notes, pages 91–108, Sep-Oct 1993.

[KSB+98] L. V. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2 : Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 1998.

[LBB97] K. Langendoen, R. Bhoedjang, and H. Bal. Models for asynchronous message handling. *IEEE Concurrency*, pages 28–38, Apr 1997.

[LTG92] D. Lischinski, F. Tampieri, and D. P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, Nov 1992.

[LTG93] D. Lischinski, F. Tampieri, and D. P. Greenberg. Combining hierarchical radiosity and discontinuity meshing. In *Computer Graphics Proceedings*, ACM SIGGRAPH, pages 199–208, Aug 1993.

[PLC95] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing'95 (CD-ROM)*, 1995.

[PRR96] A. Podehl, T. Rauber, and G. Runger. Scalability and granularity issues of the hierarchical radiosity method. In *Euro-Par Vol. I*, pages 789–798, 1996.

[RS97] J. Richard and J. Pal Singh. Parallel hierarchical computation of specular radiosity. In James Painter, Gordon Stoll, and Kwan-Liu Ma, editors, *IEEE Parallel Rendering Symposium*, pages 59–70. IEEE, Nov 1997. ISBN 1-58113-010-4.

[SAG94] B. Smits, J. Arvo, and D. P. Greenberg. A clustering algorithm for radiosity in complex environments. In *Computer Graphics Proceedings*, ACM SIGGRAPH, pages 435–442, Jul 1994.

[Sam90a] H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, Reading, 1990.

[Sam90b] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, 1990.

[SAS92] B. Smits, J. Arvo, and D. Salesin. An importance-driven radiosity algorithm. *Computer Graphics*, 26(2):273–282, Jul 1992.

[SDS95] F. Sillion, G. Drettakis, and C. Soler. A clustering algorithm for radiance calculation in general environments. In Pat Hanrahan and W. Purgathofer, editors, *Rendering Techniques '95*, pages 196–205. Springer, NY, 1995.

[SGL94] J. P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, Jul 1994.

[SH89] R. Siegel and J. R. Howell. *Thermal Radiation Heat Transfer.* Hemisphere Publishing Corporation, Washington D.C., 1989.

[She94] G. Shea. Radiosity rendering with specular shading. Master's thesis, Kansas University, May 1994.

[SHT+95] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27:118–141, Jun 1995.

[Sil94] F. Sillion. Clustering and volume scattering for hierarchical radiosity calculations. In *Fifth Eurographics Workshop on Rendering*, pages 105–117, Darmstadt, Germany, Jun 1994.

[SP89] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3):335–344, 1989.

[SSV95] W. Sturzlinger, G. Schaufler, and J. Volkert. Load balancing for a parallel radiosity algorithm. In *High Performance Computing Symposium 95*, pages 217–228, Jul 1995.

[SWPW95] D. Stuttard, A. Worrall, D. Paddon, and C. Willis. A radiosity system for real time photo-realism. In *Computer Graphics: Developments in Virtual Environments*, ACM SIGGRAPH, pages 71–81, Jun 1995.

[VC95] C. Vinod and V. Chaudhary. Parallel hierarchical radiosity algorithms : Case study on a DSM-COMA architecture. In *Int'l Conf. on Parallel and Distributed Computing Systems*, ISCA, Sept 1995.

[WEH89] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):315–324, Jul 1989.