Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods

Thomas A. Funkhouser
Bell Laboratories *

Abstract

This paper describes algorithms that allow multiple hierarchical radiosity solvers to work on the same radiosity solution in parallel. We have developed a system based on a group iterative approach that repeatedly: 1) partitions patches into groups, 2) distributes a copy of each group to a slave processor which updates radiosities for all patches in that group, and 3) merges the updates back into a master solution. The primary advantage of this approach is that separate instantiations of a hierarchical radiosity solver can gather radiosity to patches in separate groups in parallel with very little contention or communication overhead. This feature, along with automatic partitioning and dynamic load balancing algorithms, enables our implemented system to achieve significant speedups running on moderate numbers of workstations connected by a local area network. This system has been used to compute the radiosity solution for a very large model representing a five floor building with furniture.

CR Categories and Subject Descriptors:
D.1.3 [Programming Techniques]: *Concurrent Programming - Distributed Programming;* I.3.7 [Computer Graphics]: *Three-Dimensional Graphics and Realism - Radiosity.*

1 Introduction

An important application of computer graphics is lighting simulation for architectural design. Although radiosity methods are often used for simulating illumination of building interiors, current radiosity algorithms generally are not fast enough or robust enough to handle large architectural models complete with furniture due to their large computational and memory requirements. A plausible approach for accelerating such a large computation is to partition the problem among multiple concurrent processors each of which solves a separate subcomputation. This approach is particularly attractive using a network of loosely connected workstations since this type of "parallel computing resource" is common today in many industrial and research laboratories.

In this paper, we describe a new approach to executing large radiosity computations in parallel. The key innovation is a group iterative algorithm that partitions the patches into groups, and iteratively solves radiosities for patches in each group separately on different processors in parallel, while dynamically merging updated radiosities into a single solution. The primary advantage of this approach is that different group subcomputations update separate subsets of the form factors and radiosities, and therefore they can execute hierarchical radiosity solvers concurrently with little or no contention. This feature, along with dynamic load balancing algorithms, enables our implemented system to achieve significant speedups with moderate numbers of workstations distributed on a local area network. In our implementation, no single process accesses the entire scene database, and thus we are able to compute accurate radiosity solutions for very large models.

The paper is organized as follows. The next section reviews the radiosity method and describes previous work on parallel radiosity systems. Section 3 describes the classical group iterative method and discusses how it can be applied to radiosity problems. An overview of our system organization appears in Section 4, while detailed descriptions of the partitioning and load balancing algorithms are included in Section 5. Section 6 contains results of experiments with our system. Finally, Section 7 contains a brief summary and conclusion.

2 Previous work

Radiosity methods [14] simulate diffuse global illumination by computing the amount of light arriving at each patch by emission or diffuse reflection from other patches. If each patch is composed of elements (i.e., substructured [5]), the method must solve the following linear system of equations:

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

where B_i is the radiosity of element i, E_i is the emission of element i, ρ_i is the diffuse reflectivity of element i, F_{ij} is the fraction of the energy leaving element i that arrives element j, and n is the number of elements in the scene.

The primary challenges in implementing the radiosity method are efficient computation and storage of the form factors. For each form factor, F_{ij}, a visibility calculation must be performed to determine a visibility percentage for elements i and j. This calculation must consider other patches in the scene as potential blockers, and thus accounts for the majority of the computation time in most radiosity systems.

There has been considerable prior work on parallel implementations of the radiosity method. Most of this work has been applied to the progressive radiosity algorithm [6] which has $O(n^2)$ computational complexity when solved to full convergence. Implementations for this algorithm have been described for MIMD computers [2, 3, 16, 19], SIMD computers [8], transputers [9], shared memory multi-processors [1, 7], and networks of workstations [4, 20]. Recently, a few papers

*Murray Hill, NJ 07974, funk@research.att.com

have appeared describing work on parallelizing the Monte Carlo Radiosity algorithm [10, 30]. Most current implementations require a complete description of the scene's geometry to be resident in memory on all processors, thus limiting the size of models for which they can be applied.

There has been relatively little work on parallel implementations of the hierarchical radiosity method, which is surprising at first glance since its asymptotic complexity is $O(n)$ [17]. Singh [23] implemented a parallel hierarchical radiosity solver for a shared memory multiprocessor system in which each processor was initially assigned a queue of element-element interactions to process. When a processor subdivided an element, it added new interactions for the element's children to the head of its own queue. Load balancing was achieved by task stealing – idle processors removed and processed interactions from the tail of other processors' queues. Due its communication intensive nature, this approach is not practical for a network of distributed workstations.

Zareski [31] implemented a parallel version of the hierarchical radiosity algorithm on a network of workstations using a master-slave architecture in which each slave performed patch-ray intersection calculations for a separate subset of patches in the scene. For each element-element interaction, the master process constructed a set of rays and distributed them to every slave for parallel calculation of intersections with each slave's subset of the patches. After the slaves returned their hits for each ray, the master computed form factors and updated radiosities. Speedup with this fine-grained approach was thwarted by both master processing bottlenecks and the overhead of inter-process communication, resulting in longer execution times with more processors.

There are several aspects of the hierarchical radiosity algorithm that make parallelization difficult. First, since two patches can interact at any level of their hierarchies, subcomputation times are highly variable making load balancing difficult. Second, since both shooter and receiver patches can be subdivided dynamically, reader/writer locks must be used to enforce concurrency control during updates, and deadlock avoidance/resolution must be considered. The computation required to manage concurrency control and deadlock during access to the element radiosity and mesh hierarchies can significantly reduce speedup results.

In summary, none of the previous parallel systems is fast or robust enough to compute an accurate radiosity solution for a very large building model because they suffer from at least one of the following shortcomings: 1) greater than $O(n)$ computational complexity, 2) replication of the entire model for each processor, 3) inaccuracies due to energy transfer or form factor approximations (e.g., hemi-cube artifacts), 4) limited speedup due to contention during access to shared data, and 5) communication overhead for process control.

In order to scale to very large models, a radiosity system must use an efficient matrix solution method, such as hierarchical radiosity. The algorithm must be partitioned into separate concurrent subcomputations that each access a small subset of the model. In order to scale to many processors, the separate subcomputations must read/write separate regions or copies of the model to avoid slowdowns due to contention. Finally, the granularity of parallelism must be coarse enough to allow execution with minimal overhead for communication between participating processors. The design and implementation of a parallel radiosity system meeting these criteria is the topic of this paper.

3 Group Iterative Methods

The radiosity method must solve a linear system of equations represented by the row-diagonally dominant interaction matrix shown in Figure 1. Group iterative methods partition the B_i variables into groups, and rather than just relaxing one variable at a time, they relax an entire group during a single step [13, 29]. Gauss-Seidel group iteration relaxes each group using current estimates for B_js from other groups, while Jacobi group iteration uses B_js from other groups updated at the end of the previous complete iteration through all groups.

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

Figure 1: The radiosity matrix equation.

Application of Jacobi group iteration in the radiosity domain can be interpreted as partitioning elements into groups, and then iteratively "gathering" radiosity to elements using current radiosities from elements in the same group and radiosities from the end of the previous complete iteration for elements in other groups. Elements within the same group bounce energy back and forth to convergence during each iteration, while elements in different groups exchange energy only once per iteration (see Figure 2).

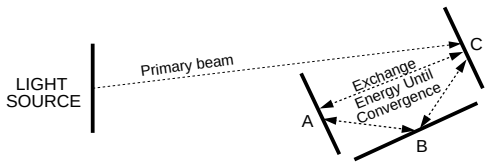

Figure 2: One relaxation step for group "ABC".

There are several advantages to the group iterative approach for large radiosity problems, particularly with regards to parallel processing. First, each group "gathering" step updates radiosities only for the elements in its group, which is advantageous for concurrency control when compared to "shooting" algorithms that update radiosities for all elements in each step [3]. Second, with Jacobi methods, updates to the radiosity values of elements in each group depend only upon radiosity values copied at the end of the previous iteration, and do not require access to current radiosity values for elements in all groups. This property allows multiple radiosity solvers to execute concurrently on different groups, with each solver updating a separate copy of the radiosity values without readers/writers contention. It then becomes practical to use efficient, yet complicated, radiosity algorithms, such as Hierarchical Radiosity [17], to solve each group subproblem. Finally, group methods exhibit better cache coherence than element-by-element methods [13] since links between patches in the same group can be reused several times as the group is solved to convergence. This feature is particularly important for radiosity problems whose form factor matrices do not fit in memory all at once.

In this paper, we describe the design and implementation of a radiosity system based on group iterative techniques that uses multiple concurrent hierarchical radiosity solvers. For each iteration, the system automatically partitions the patches describing a scene into groups and executes

hierarchical radiosity solvers to compute converged radiosity solutions for separate groups on separate processors using separate versions of the model in parallel. Throughout the computation, updated versions of the element radiosities are copied into a master scene database for later use by other processors. Since hierarchical substructuring and form factor calculations are performed for different groups in parallel on the separate processors accessing separate copies of the model, we can accelerate overall computation times due to parallelism with little or no contention. Since coordination of processes is performed at a coarse-grained level (groups), relatively little communication is required between processes. As a result, significant speedups are possible for moderate numbers of processors. Furthermore, since each processor must store only the working set for computations for one group at a time, the approach scales to support very large models.

4 System Organization

Our system is organized in a Master-Slave configuration with one master and P slaves running concurrently on separate processors. The slaves are used to execute radiosity computations, while the master performs dynamic load balancing and data distribution. All processes maintain independent (partial) copies of the scene database, and slaves communicate with the master only via TCP messages. This organization allows distribution across loosely-coupled workstations without shared memory, or even shared disks.

4.1 Flow of Control

The flow of control between the master and slave processes is shown in Figure 3. The master iteratively relaxes groups until convergence. For each "master iteration," the master partitions patches of the scene database into groups, and then dynamically distributes the groups to slaves one at a time for group relaxation computations. These automatic partitioning and scheduling algorithms are the focus of this paper, and are described in detail in the following section. This section describes the organization of the system in which these algorithms execute.

The master starts by spawning P slave processes (usually on remote computers) and opens a TCP socket connection to each of them. It uses the `select` UNIX system call to detect messages from multiple slaves. Whenever a slave, S_i, indicates it is ready, the master selects a group, G, from a list of candidate groups waiting to be processed during the current master iteration. Next, it downloads to S_i all patches potentially visible to any patch in group G (i.e., the "working set" for G). After the downloads have completed, the master sends slave S_i a message indicating that it can begin its radiosity computations for group G. While S_i relaxes group G to convergence, the master continues servicing other slaves. After slave S_i finishes its computation, it sends updated radiosity values back to the master for use in the current or future iterations.

Each slave runs asynchronously on a separate processor under the guidance of the master process as shown on the right side of Figure 3. When a slave receives a *download* message from the master, it updates its local copy of the patches it receives, waits for a *compute* message from the master, and then invokes a hierarchical radiosity solver to gather radiosity to all patches in group G until convergence.

The radiosity solver is based on the hierarchical (wavelet) radiosity system described in [15, 17, 27]. Although its details are not the focus of this paper, it is important to note that it stores its evolving solution in a disk-resident database and loads into memory only the data required for the current subcomputation. It manages a fixed size, memory-resident, LRU cache to store the most recently used elements and links (form factors) in hopes that they will be used again before they are discarded. As computation of the form factors is the most costly component of the system execution, effective management of this cache is critical to avoiding costly recomputation or re-loading from disk. This feature of the slave solver is advantageous for the group iterative approach. Since gathers are performed multiple times to the elements of the same group in succession, the group algorithms exhibit far greater cache coherency, and in our case, we are often able to re-use links computed for two elements multiple times before they are discarded from the cache. In contrast, effective cache management would be very difficult with classical gather algorithms that make successive sweeps over all patches in the entire database.

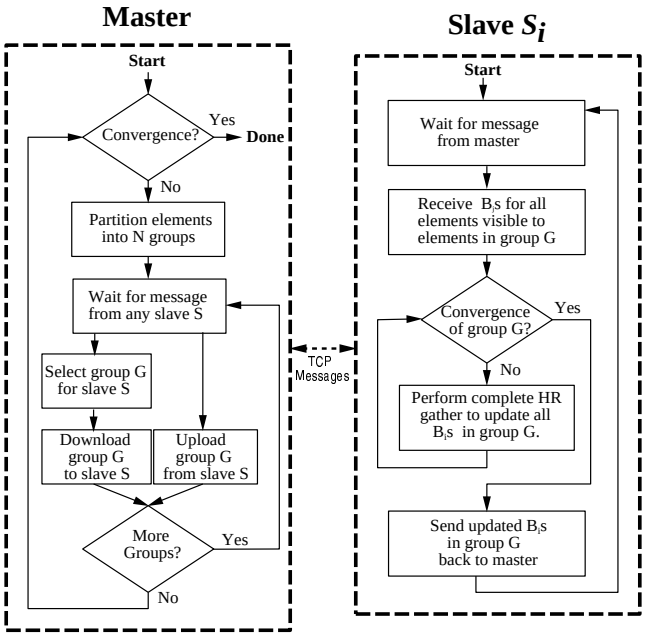

Figure 3: Master-slave flow of execution.

After the radiosity computation for group G has been completed, the slave writes into its local cache updated versions of all patches in G, including the refined hierarchical element meshes and radiosities for each patch, and sends to the master an *upload* message containing a packed representation of these patches. This updated version of G is merged into the master scene database and can potentially be downloaded to other slaves for later computations.

Note that communication between master and slave occurs only three times for each group iteration: 1) to download patches to the slave at the beginning of an iteration, 2) to invoke a radiosity computation, and 3) to upload patches from the slave at the end of an iteration. This coarse-grained approach to parallelism is important. Other efforts to parallelize the radiosity method with a master-slave organization have found master processing to be a bottleneck, and communication overhead has diminished speedup results significantly using relatively few slave processors. Our strategy is to design a system in which a master coordinates execution of the slaves, but at a very coarse granularity, with very infrequent communication.

4.2 Data Distribution

The scene description is initially available only to the master. It is stored in a database containing patches represented by quad-trees of elements with diffuse reflectivity, radiosity, and emission attributes. The patches are stored in the scene database arranged in clusters specified by the modeler at scene creation time. The scene database also contains precomputed cluster-to-cluster visibility information. The cluster visibility calculation is performed off-line using the algorithms described in [26] and generates a list for each cluster indicating which other clusters are potentially visible to it – i.e., not occluded by a wall, ceiling, or floor. Although clustering and visibility techniques are an important research area, and essential to the efficient execution of our radiosity system, these topics are not addressed in this paper. See [21, 22, 24, 26] for further information.

Since only the master has access to the complete scene database, it must download portions of the database (i.e., potential working sets) to slaves during execution. We define the potential working set for a group G to be all the patches, including their element meshes and radiosities, that are visible to any patch in G. This definition of the working set is a conservative over-estimate of the set of data the slave may need access to during radiosity gather operations for any group. Since patches in occluded clusters cannot exchange energy directly, we can use the precomputed cluster-to-cluster visibility information of the scene database to compute the potential working set efficiently.

During execution, the master keeps an index of which clusters have already been downloaded to each slave. It traverses the cluster-to-cluster visibility lists for all clusters containing patches in group G, checking whether the potentially visible cluster, C, is already up-to-date on the slave, S_i. If not, it reads from the disk-resident scene database all the data describing patches in C, including the hierarchical mesh of elements with radiosities. It packs this data into a buffer and performs a write to the TCP socket for slave S_i. Finally, it marks slave S_i up-to-date for cluster C in its index, and continues checking for other potentially visible clusters to download. Note that all patches of a cluster are required to be in the same group, making this download processing somewhat more efficient. After all clusters in the working set of G are up-to-date on slave S_i, the master sends a short message indicating which clusters belong to group G, and directing it to gather radiosity to all patches in those clusters to convergence.

After the slave has updated the radiosities for all patches in group G to convergence, it sends an upload message to the master with complete updates for every cluster C in group G. The master writes these updates back to a new version of cluster C in its disk-resident database. It then marks clusters in G out-of-date for all slaves except S_i, causing them to be freshly downloaded for subsequent gather operations for clusters visible to G. With this concurrent "copy-update-replace methodology," our system does not truly execute either the Jacobi group or Gauss-Seidel group iterative method, as it is indeterminate whether the old or updated copies of a group's variables will be used during each group relaxation step. Proof of convergence with this optimization is shown in the following section.

The data distribution features of our system are important for scaling to support computations with very large models. The master stores only the scene database header information in main memory (generally less than 20MB), while the clusters, patches, and elements reside on disk. Each slave receives and stores only the subset of the scene database required for its computation, avoiding full replication of the entire database on any processor as is required by most other parallel radiosity systems.

4.3 Convergence Proof

Proof of convergence of our parallel group iterative method can be shown by comparison to the standard sequential group Jacobi method, which is known to converge [29]. Consider splitting the matrix A (as in $Ax = b$) into $A = D - L - U$ where D has blocks along the diagonal, L has the opposites of element below D, and U has opposites of elements above D. For the radiosity equation, A is monotone (i.e., it has non-negative elements along the diagonal and non-positive elements elsewhere), D is monotone, $L \geq 0$, and $U \geq 0$.

The standard group Jacobi method iterates according to the following equation:

$$x_{k+1} = I_J x_k + D^{-1} b$$
$$where\ I_J = D^{-1}(L + U)$$

whereas our modified group method iterates using some variables updated in the current iteration and some updated in the previous iteration:

$$x_{k+1} = I_M x_k + (D - L_1^k)^{-1} b$$
$$where\ I_M = (D - L_1^k)^{-1}(L_2^k + U),$$
$$L = L_1^k + L_2^k, L_1^k \geq 0,\ and\ L_2^k \geq 0$$

We can show that our modified group method converges if the error is reduced during each iteration. Since

$$\frac{||x_{k+1} - x||}{||x_k - x||} \approx \rho(I_M)$$

where $\frac{||x_{k+1}-x||}{||x_k-x||}$ is a suitable vector norm, and $\rho(I_M)$ is the spectral radius of I_M, convergence is guaranteed if $\rho(I_M) \leq \rho(I_J) < 1$. We prove $\rho(I_M) \leq \rho(I_J)$ using corollary 5.6 on page 125 of Young [29]:

> "Let A be a monotone matrix and let $A = Q_1 - R_1$ and $A = Q_2 - R_2$ be two regular splittings of A. If $R_2 \leq R_1$, then $\rho(Q_2^{-1} R_2) \leq \rho(Q_1^{-1} R_1)$."

A regular splitting of A is one in which $A = Q - R$ where $Q^{-1} \geq 0$ and $R \geq 0$. For the standard group Jacobi iteration method, let $A = Q_1 - R_1$ where $Q_1 = D$ and $R_1 = L + U$. Note that $Q_1^{-1} \geq 0$ since D is monotone and $R_1 \geq 0$ since $L \geq 0$ and $U \geq 0$. For our modified group iteration method, let $A = Q_2 - R_2$ where $Q_2 = D - L_1^k$ and $R_2 = L_2^k + U$. Note that $Q_2^{-1} \geq 0$ since $D - L_1^k$ is monotone. Also note that $R_2 \geq 0$ since $L_2^k \geq 0$ and $U \geq 0$, and $R_2 \leq R_1$ since $L \geq L_2^k$.

Applying corollary 5.6 and convergence of the standard group Jacobi method, we see that $\rho(I_M) < 1$ and the modified group iterative method must converge:

$$\rho(I_M) = \rho(Q_2^{-1} R_2) \leq \rho(Q_1^{-1} R_1) = \rho(I_J) < 1$$

5 Parallel Programming

A general strategy for parallel programming is to decompose a computation into a set of independent subcomputations, and then to distribute the subcomputations for execution in parallel on available processors. The important issues are to find an appropriate decomposition (i.e., partition patches into groups), and to schedule execution of the subcomputations effectively (i.e., load balancing). These issues are addressed in detail in this section.

5.1 Group Partitioning

Goals and Strategies

Based on intuition derived from experimentation with our system, we have developed the following set of guidelines that constrain our automatic partitioning algorithms: 1) the number of groups, N, should be bounded from below so that there are guaranteed to be enough groups to schedule effectively on P slave processors (e.g., $N > 8P$); 2) each group should be large enough that the time required to distribute its computation to a slave is not more than it would have been to perform it locally on the master; and 3) each group should be small enough that the links for radiosity updates to all elements in the group fit in a slave's memory-resident cache so that they may be re-used over and over again without recomputation as the group is solved to convergence. We combine these constraints with the goals of maximizing intra-group form factors while minimizing inter-group form factors to form the basis of our partitioning algorithms.

For practical purposes, we consider only partitionings in which all patches of a cluster are assigned to the same group. This restriction simplifies the partitioning algorithms, and aids execution of the data distribution algorithms during execution of our radiosity system, as described in the previous section.

Conceptually, we address the cluster partitioning problem as a computation on a *form factor graph* in which each node in the graph represents a cluster, and each edge represents an estimate of the form factor between its nodes' clusters (a simple form factor graph is shown in Figure 4). With this formulation, we state the cluster partitioning problem as follows: assign nodes of the form factor graph to groups such that the cumulative weight of edges between nodes in the same group divided by the cumulative weight of all edges is maximal.

Unfortunately, this problem is equivalent to the Graph Bisection Problem [12], which is known to be NP-complete. However, we have developed two automatic algorithms that find approximate and useful solutions in polynomial time. The first algorithm, called the *Merge Algorithm*, starts by assigning each cluster to a separate group and then iteratively merges groups. Conversely, the second algorithm, called the *Split Algorithm*, starts by assigning all clusters to the same group and then recursively splits groups. Either algorithm can be used to construct groups, or the algorithms can be applied successively to iteratively refine groups.

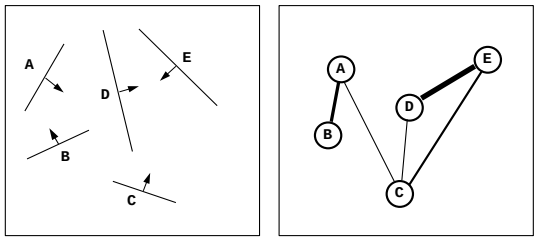

Figure 4: Simple scene (left) with its form factor graph (right). Edge thickness represents form factor magnitude.

Merge Algorithm

The *Merge Algorithm* operates on an augmented version of the form factor graph in which nodes represent groups rather than clusters. In this augmented graph, the edge between two nodes representing groups A and B has weight equal to the sum of the form factors between all combinations of clusters in groups A and B. Initially, a graph is created with one node for each cluster. For the purposes of constructing this graph, an edge weight is set to zero (or the edge is not created at all) if two clusters are known to be occluded from one another (as determined by a lookup in the precomputed cluster-to-cluster visibility information stored in the scene database). Otherwise, the form factor, F_{AB}, from one cluster A to another cluster B is estimated as the solid angle subtended by a disk representing cluster B [28]:

$$F_{AB} = r^2/(d^2 + r^2) \quad (2)$$

where d is the distance between A and B, and r is the radius of a sphere bounding B. This approximation is an overestimate that does not consider individual patch orientations and assumes that A is entirely visible to B.

Once the form factor graph has been constructed, the Merge Algorithm iteratively merges groups (nodes of the graph) until no further combinations are possible within the following constraints: 1) the number of groups is greater than a user specified minimum, *MinGroups*, and 2) the estimated number of links for any group with more than one cluster is below a user specified maximum, *MaxLinks*. By default, *MaxLinks* is arbitrarily set to be $1.25 * TotalLinks/MinGroups$, where *TotalLinks* is the sum of the link estimates for all groups.

The key challenge for implementation of the Merge Algorithm is selecting two appropriate groups to merge during each step of the algorithm. We take a greedy approach. The pair of groups, A and B, is chosen whose merger causes the greatest increase in the ratio of intra-group edge weights divided by the total of all edge weights. If the merger of these groups meets all constraints, they are combined into one. During the merge operation, edges from A and B to other nodes are replaced by ones to the new merged node. The weight of this new edge is the sum of the weights of the edges it replaces (see Figure 5). The algorithm repeatedly merges groups until it can no longer find any pair of groups to merge legally, or the solution cannot be improved. In the worst case, when all clusters are visible to one another, the algorithm is bounded by $O(N^2 logN)$. However, in situations such as building interiors, where visibility sets are usually of constant size, the average execution time for the merge algorithm is $O(N logN)$.

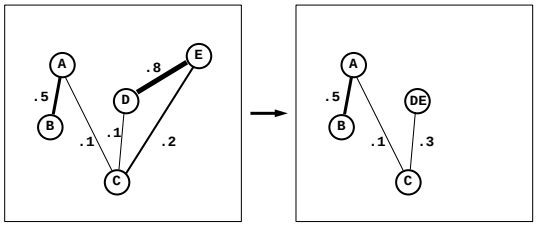

Figure 5: Merge operation for nodes 'D' and 'E'.

Split Algorithm

The *Split Algorithm* uses a strategy that is the converse of the Merge Algorithm. It starts with all clusters assigned to a single group and then recursively splits groups until further splits do not improve the solution. This algorithm can be interpreted as a recursive binary partitioning of the form factor graph.

During each step of the algorithm, our goal is to choose an appropriate partition of one group into two new ones that meet all size constraints and have minimal inter-group form factors. We use geometric split heuristics originally developed for construction of spatial subdivisions for use in visibility determination (e.g., BSP trees [18]). Specifically, we partition the model along planes aligned with "major occluding" polygons of the model (see [25] for details). As the model is split recursively by these planes, clusters are assigned to groups depending on whether their centroid lies above or below the splitting plane (see Figure 6). This process is applied recursively until no groups can be split within minimum group size constraints, or until no further "major occluder" polygons can be found. The algorithm runs in $O(NlogN)$. If split planes are chosen appropriately (i.e., such that the cumulative form factors between clusters on separate sides of the plane are small), it generates a partitioning with little exchange of energy between groups during a radiosity simulation.

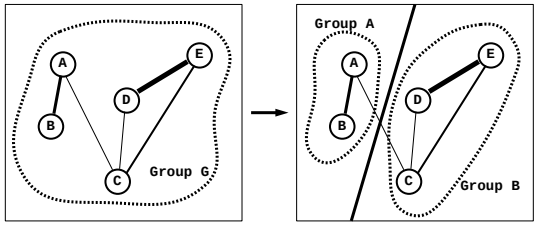

Figure 6: Split operation creating groups 'A' and 'B'.

Figure 7 shows two sets of 16 groups constructed using the Merge and Split algorithms, respectively, for a one floor building model comprising 1667 clusters. (clusters are shaded based upon which group they were assigned). Using the Merge algorithm, groups tend to be formed from clusters that are visible to each other (e.g., offices across hallways), whereas groups tend to be formed from clusters that are nearby each other using the Split algorithm (e.g., neighboring offices).

Merge Algorithm Split Algorithm

Figure 7: Groups formed by the merge and split algorithms.

5.2 Scheduling

Load balancing is a primary concern in any parallel system. Our goal is to schedule group radiosity subcomputations on slaves in a manner that maximizes the rate of convergence to an overall solution. Unfortunately, this *Multi-Processor Scheduling Problem* is NP-Complete since each subcomputation is non-preemptable, task execution times are highly variable, and workstations may have different performance capabilities [11]. In this section, we describe our approximation algorithms for scheduling and load balancing.

First-Fit Decreasing Algorithm

A common scheduling strategy for minimizing the total completion time for a set of tasks run on multiple processors is to select tasks in order of their expected execution times, largest to smallest. This strategy is called the *First Fit Decreasing* (FFD) algorithm [11]. The idea is to schedule the large tasks first so that there is less chance that their execution times will extend beyond the last execution time of any other task.

We have applied this principal in our radiosity system. The difficult challenge is to predict in advance how long a radiosity computation for a group will take. We estimate the relative compute time for a transfer of radiosity from one cluster A to another cluster B by F_{AB}. This estimate is based on the observation that slave compute times are dictated by the number link evaluations (ray-patch intersections), which is determined by the errors in computed element-element form factor estimates, which in turn are roughly correlated to form-factors. In order to estimate the computation time for gathers to a group of clusters, G, we sum estimated computation times for all cluster pairs in which at least one of the two clusters is in G, and the clusters are known to be at least partially visible to one another via the form factor graph.

To execute the FFD algorithm, the master sorts groups according to computation time estimates as they are constructed. Then, groups are simply assigned in FFD order as slaves become available during execution.

Working Set Algorithm

The general principal of minimizing total completion time for a set of independent subcomputations is not enough to guarantee a fast convergence rate for our radiosity system. We must also consider factors affecting data download performance, duplicate calculation, and energy distribution. These issues are particularly important because each slave maintains a local cache of data containing element radiosities and links previously computed. The history of which groups a slave has previously processed affects the download time and the energy distribution rate for the current computation. These issues are likely even more important for a system utilizing bi-directional links (our system creates unidirectional links) in which case re-use of inter-group links could be a significant scheduling consideration.

We have developed a dynamic scheduling algorithm that considers data download factors when scheduling group computations on slaves. The *Working Set* (WS) algorithm uses a heuristic that is designed to assign groups to slaves for which their working set has already been downloaded. Each time a slave S_i becomes available, it considers groups remaining to be processed during this iteration. For each candidate group, G, it computes the percentage of the clusters visible to any cluster in G that are already resident on S_i. It then subtracts from this value the percentage of clusters visible to G that are not resident on S_i, but are resident on some other slave. This latter factor helps to keep the visibility sets of groups assigned to different slaves separated. The difference between these two percentages forms the heuristic that the Working Set algorithm uses to choose the best group for each slave dynamically as the system executes.

Combined Scheduling Algorithm

The methodologies of the FFD and WS algorithms can be combined. We generally use a combined scheduling algo-

rithm (FFD-WS) that dynamically chooses a group as each slave becomes available according to the WS heuristic subject to the constraint that every group must be scheduled within "*delta*" slots of its position in FFD order. This algorithm is equivalent to the FFD algorithm if $delta = 1$, and it is equivalent to the WS algorithm if $delta = \infty$. Otherwise, if $1 < delta < N$, we hope to realize the advantages of both the FFD and WS approaches.

6 Results and Discussion

In order to test the effectiveness of the group iterative approach for solving large radiosity problems in parallel, we executed a series of experiments with our system using different group partitioning and load balancing algorithms. During these experiments, we used up to eight Silicon Graphics slave workstations, each with a 150MHz R4400 processor and at least 80MB of available memory, 32MB of which was available for caching links. The workstations were spread over two separate local area networks and did not share disk files. Unless stated otherwise, the *Merge Algorithm* was used to construct 256 groups, and the *FFD-WS* algorithm was used with $delta = 16$ for dynamic load balancing. In all experiments except the one described in Section 6.1, the master process performed two complete iterations in which a slave gathered to every patch in each group twice with a moderately fine error tolerance. During the initial slave iteration, patches gathered radiosity only from the lights.

Our test model in every experiment was the computation of a radiosity solution for one unfurnished floor of the Soda Hall building model. This test model contained 6,418 patches in 1,667 clusters, 242 of which contained only emissive patches. The total area of all surfaces was 10,425,645 square inches. Although this test model was not particularly complex, it was useful for experimentation. With a larger model, it would have been impractical for us to investigate algorithmic trade-offs by performing many executions of the radiosity solver with different parameters.

6.1 Group Iteration Results

We first compared the performance of the group iterative method to traditional iterative methods (independent of parallel processing) by computing the radiosity solution for our test model using a single processor both with and without grouping of patches. During the first test, patches were not grouped, and 4 traditional Gauss-Seidel iterations were made over all patches. During the second test, patches were partitioned into 256 groups by the Merge algorithm. Then, three Gauss-Seidel group iterations were made over all groups, during which every patch in a group gathered radiosity twice (groups were not solved to full convergence during each step). During the test without grouping, every patch gathered radiosity from every other patch four times. In contrast, during the test with grouping, patches gathered radiosity six times from patches within the same group, but only three times from patches in other groups. Plots of transfer rates measured during these tests are shown in Figure 8. Circles on the plots indicate the end of a compete sweep through all variables in each test.

Even without parallel processing, the group iterative method out-performed the traditional approach during this experiment. The performance difference was mostly due to the fact that the group method more effectively made use of links and patches cached in memory by the solver. As described in Section 4, the solver maintained LRU memory

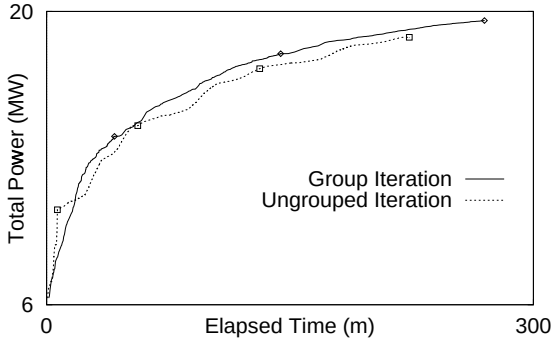

Figure 8: Transfer rates for grouped/ungrouped iteration.

resident caches of links and patches. Patches that did not fit in the cache had to be flushed to disk, while links that did not fit in the cache were discarded and later recomputed. During this experiment, although the total amount of storage required for links exceeded the cache limit (32MB), the maximum working set for any group did not. As a result, since the group method cycled over patches in each group multiple times in succession, it was often able to re-use previously computed links (45% of the time). In contrast, the traditional method executed a worst-case access pattern for the LRU cache, making complete sweeps through all patches in succession, and thus was not able to re-use any links.

6.2 Partitioning Results

We next studied the effects of different group partitioning algorithms by executing a sequence of tests with 8 slaves using the following methods to partition clusters into 256 groups:

- **Merge:** Groups were constructed using the Merge Algorithm with $MinGroups = 256$.
- **Split:** Groups were constructed using the Split Algorithm partitioning on floors, ceilings, and walls of the building model with $MaxGroups = 256$.
- **Region:** Clusters were assigned to groups based on the (x,y) coordinates of their centroids in a 16x16 grid.
- **Random:** Clusters were assigned to groups randomly.

Figure 9 contains plots of transfer rates measured during these tests. The system converged fastest using partitions generated automatically with the Merge and Split Algorithms. This is due to the fact that these algorithms combined clusters into the same group based on estimated form factor and proximity relationships. During every test, each patch gathered radiosity a total of four times – two iterations in a slave for each of two master iterations. This

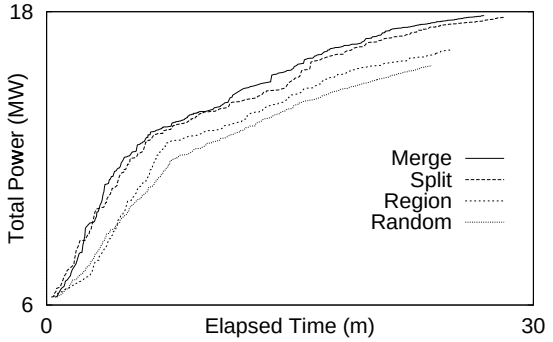

Figure 9: Transfer rates for different partitioning algorithms.

means that energy was distributed with four reflections between clusters in the same group, while only two reflections occurred between clusters in different groups. As expected, the performance of the group iterative approach was better using partitions with larger intra-group form factors.

6.3 Granularity Results

We studied the effects of group granularity by measuring system performance using 8 slaves for tests with groups of different sizes. Using the Merge Algorithm, we executed tests with the clusters partitioned into 32, 128, 256, and 1,425 groups. The test with 1425 groups represents construction of a separate group for each cluster containing at least one reflective patch.

Plots of transfer rates measured in each test appear in Figure 10. We found that the advantage of the group iterative method is reduced if groups are very small since there is little opportunity to re-use links computed for intra-group radiosity transfers. On the other hand, if we used just a few large groups, the data required for all intra-group links exceeded a slave's cache capacity for some groups, reducing the effectiveness of the cache. Also, it was more difficult to schedule a relatively few, large subcomputations across available slave processors in order to achieve the best possible completion times. During our experiments, tests performed best with 256 groups that roughly corresponded to the small, convex regions of the model. This result depends on a variety of factors, including the size of link caches in slaves and the variability of group sizes.

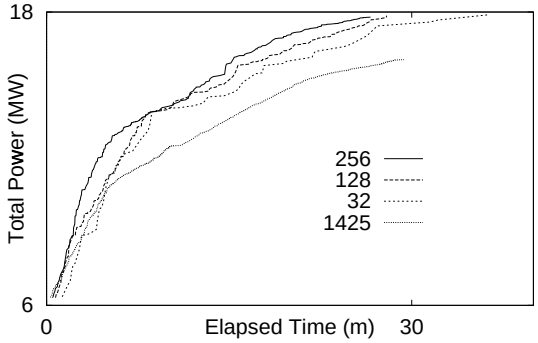

Figure 10: Transfer rates for different group granularities.

6.4 Scheduling Results

We investigated load balancing and scheduling effects by executing a series of tests using 8 slaves with different scheduling algorithms:

- **FFD:** Groups were assigned to slaves in FFD order.
- **WS:** Groups were assigned to available slaves dynamically to minimize the WS heuristic.
- **FFD-WS:** Groups were assigned to slaves dynamically using the FFD-WS algorithm with $delta = 16$.

From statistics measured during these tests, we found that the scheduling factors impacting convergence rates most were: 1) master-slave download times, and 2) slave idle times (particularly at the end of each iteration). As expected, the master spent the least amount of time downloading data to slaves during tests using the WS algorithm (98 seconds). The advantages of the WS approach can be seen in Figure 11, in which all 256 groups are shaded according to which of the 8 slaves they were assigned during tests using the FFD and WS algorithms. The coherence of the working sets assigned to slaves using the WS algorithm allows the system to minimize data downloads and maximize energy transfers.

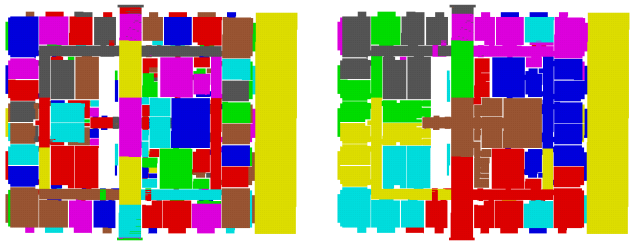

Figure 11: Visualization with groups shaded by slave.

Unfortunately, the test using the WS algorithm also spent the most amount of time waiting for the last slave to finish at the end of each iteration (547 seconds). In particular, one very large group computation was postponed until the very end of the second iteration, causing the master and seven of the slaves to sit idling while the eighth slave finished its computation for that group. In contrast, the FFD algorithm spent a small amount of time waiting for the last slave at the end of each iteration (13 seconds), but it spent the most time downloading data to slaves (248 seconds).

The trade-offs between scheduling to minimize downloads and scheduling to minimize time waiting for the last slave can be seen in Figure 12, which shows a vertical elapsed time-line for each of the 8 slaves during tests with the FFD, WS, and FFD-WS algorithms. Every distinct vertical bar segment represents radiosity computation for one group on one slave. Using the FFD algorithm, the execution time predictor does fairly well, and longer tasks are generally scheduled earlier in each iteration (the first of the two master iterations ends approximately 1/3rd of the way up the time-line). However, because the master spends more time downloading data to slaves (synchronously), there are more frequent and longer periods during which a slave is waiting for the master (blank spaces between vertical bars). Using the WS algorithm, although download times are far less (intra-bar gaps are smaller), the computation for one very large group was scheduled near the end of iteration 2 (on Slave 2) causing the master and all other slaves to wait for it to complete. The combined FFD-WS algorithm seemed to achieve a good combination of download times (191 seconds) and wait times (8 seconds), and thus converged most rapidly.

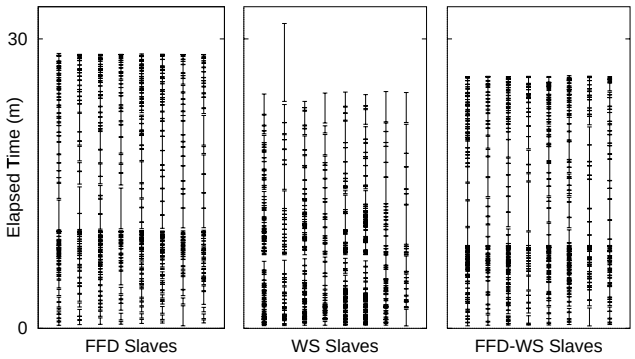

Figure 12: Slave compute time (solid) vs. wait time (blank).

6.5 Speedup Results

Finally, we executed an experiment to determine how much speedup is possible with our system via parallel processing. During this experiment, we solved the one floor test model four times using 1, 2, 4, and 8 slaves, respectively. A plot of speedup for increasing numbers of slave processes is shown in Figure 13.

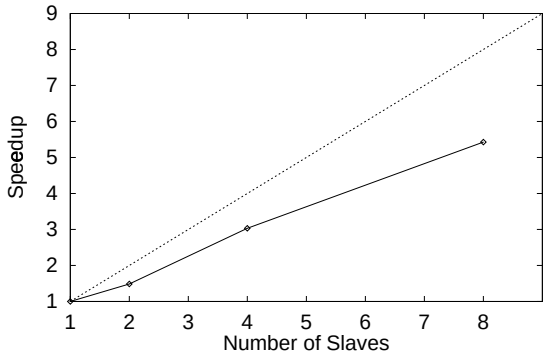

Figure 13: Transfer rate speedup for 1, 2, 4, and 8 slaves.

For up to 8 slave processors, the system maintains a 65-75% speedup. The speedup is less than 100% due to the synchronous master-slave communication model of our system. Although the group iterative approach provides a relatively coarse granularity of parallelism, the master communicates with slaves synchronously in our current implementation – i.e., it can only talk to one slave at a time. As a result, if two slaves finish a subcomputation and become ready for further processing at the same time, one must wait while the master exchanges data with the other. The impact of this effect is determined by the likelihood that a slave will finish a subcomputation while the master is processing data for another slave. Although this likelihood grows with the number of slaves, it is also affected by the relationship of time required for master processing of downloads/uploads versus the time required for slave processing of a group radiosity subcomputation. For solutions in which the slave radiosity subcomputations are longer relative to the data distribution times, speedup results are better.

The speedup bottleneck resulting from our current synchronous communication model with a centralized master can be mitigated somewhat by enhancing the master to use asynchronous I/O protocols or by switching to communication protocols in which slaves transfer data among themselves under master coordination. Based on our preliminary results, we are optimistic that the group iterative approach provides coarse enough granularity that our master-slave system can scale to large numbers of slave processors with the addition of enhanced communication methods. Unfortunately, we have not yet implemented these improvements, and do not currently have access to enough workstations to determine the absolute limits of our current system.

The speedup experiments point out an interesting trade-off of our parallel group iterative approach. On one hand, when more slaves compute concurrently, we are able to evaluate more element interactions in less time. On the other hand, since updated radiosity values are distributed from the master to a slave workstations only after they have been uploaded from other slaves, Gauss-Seidel group iteration is achieved only during tests with one slave. In contrast, if all groups were scheduled simultaneously on different slaves, the system would perform a true Jacobi group iteration. As more slaves are added to the system, the system more closely resembles Jacobi iteration since more and more computations are performed with copies of radiosity values last updated at the end of the previous iteration. Further research is required to investigate the impact of this effect.

6.6 Practical Results

As a final test, we computed a radiosity solution for a very large model using the system described in this paper. The model represents five floors of a large building with approximately 250 rooms containing furniture. It was constructed with 14,234 clusters comprising 280,836 patches, 8,542 of which were emitters and served as the only light sources. The total area of all surfaces was 75,946,664 square inches. Three complete iterations were made through all patches using an average of 4.96 slave processors in 168 hours. The entire computation generated 7,649,958 mesh elements and evaluated 374,845,618 element-to-element links.

During this execution, the master spent 0.8% of its time constructing and scheduling groups, 4.4% downloading data to slaves, 2.6% uploading results from slaves, and 89.0% waiting for slaves. The slaves spent 0.1% of its time downloading data from the master, 0.1% uploading results to the master, 79.1% updating radiosities, and 5.0% waiting for the master. Although it was not practical for us to solve this model using a single processor for comparison, we estimate the speedup due to parallelism as the time spent performing radiosity computations in slaves divided by the elapsed time, which was 3.9 in this case, or 79% of linear speedup.

Figure 14 shows renderings of this large radiosity solution from various viewpoints captured during an interactive walkthrough. Outlines around mesh elements are included in the bottom-right image for detailed inspection. Note the adaptive refinement of elements in areas of partial visibility (e.g., on the floor near the legs of tables and chairs) due to hierarchical radiosity meshing. To the author's knowledge, this model is the most complex for which a radiosity solution has ever been computed.

7 Conclusion

This paper describes a system for computing radiosity solutions for very large polygonal models using multiple concurrent processes. A master process automatically partitions the input model into groups of patches and dynamically schedules slave processes which execute independent hierarchical radiosity solvers to update the radiosities of patches in separate groups. During experiments with this system, uniprocessor group methods out-performed traditional methods due to improved cache coherence, while multi-processor group methods achieved further speedups of 65-75% using up to 8 slave workstations.

We have found that the implementation and analysis of a distributed approach to the radiosity problem requires careful consideration of group partitioning, data distribution, and load balancing issues. Coarse-grained parallel execution using multiple separate copies of a shared database allows multiple processors to execute concurrently with little contention or synchronization overhead. However, since updates to the shared database are executed with coarse granularity, many of the subcomputations may be performed using out-of-date database values, potentially reducing the convergence rate.

The conflicting goals between computing in parallel versus computing with the most up-to-date data results in an

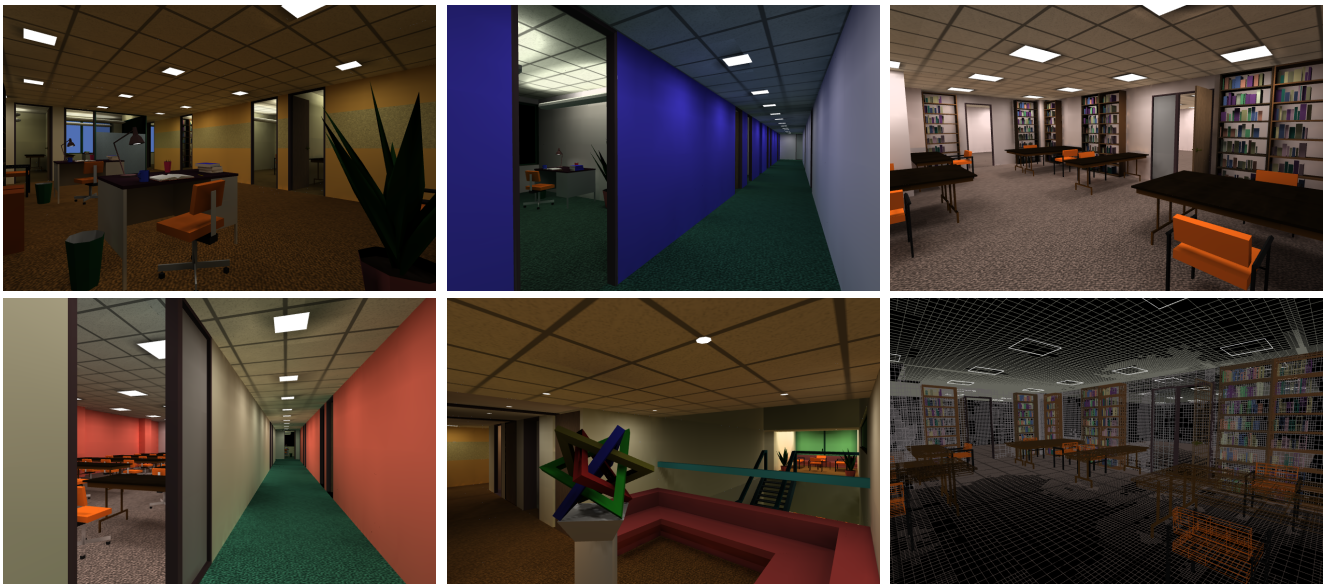

Figure 14: Images captured during an interactive walk through one large radiosity solution.

interesting trade-off whose resolution is affected by a multitude of factors, including the size of workstation memories, the size of working sets, the speed of the network, and so on. We believe that examining issues in parallel execution for large computations distributed over a network is an interesting research area that will become more and more important as networked computing resources become more and more prevalent.

Acknowledgements

The author thanks Roland Freund and Wim Sweldens for developing the convergence proof appearing in this paper. I am also grateful to Pat Hanrahan, Peter Schroder, and Stephen Gortler for their helpful insights and discussion, and to Seth Teller and Celeste Fowler for their efforts building the original radiosity system at Princeton. Finally, special thanks to Carlo Séquin and the UC Berkeley Building Walkthrough Group for building the model of Soda Hall and for getting me started on this research project.

References

[1] Baum, D., and Winget, J. Real Time Radiosity Through Parallel Processing and Hardware Acceleration. Computer Graphics (1990 Symposium on Interactive 3D Graphics), 24, 2, 67-75.

[2] Bouatouch, K., and Priol, T. Data Management Scheme for Parallel Radiosity. Computer-Aided Design, 26, 12, December, 1994, 876-883.

[3] Chalmers, A, and Paddon, D. Parallel Processing of Progressive Refinement Radiosity Methods. Second Eurographics Workshop on Rendering, Barcelona, Spain, May, 1991.

[4] Chen, S.E. A Progressive Radiosity Method and its Implementation in a Distributed Processing Environment. Master's Thesis, Cornell University, 1989.

[5] Cohen, M., Greenberg, D., Immel, D., and Brock, P. An Efficient Radiosity Approach for Realistic Image Synthesis. IEEE Computer Graphics and Applications, 6, 3 (March, 1986), 25-35.

[6] Cohen, M., Chen, S., Wallace, J., and Greenberg, D. A Progressive Refinement Approach to Fast Radiosity Image Generation. Computer Graphics (Proc. SIGGRAPH '88), 22, 4, 75-84.

[7] Drettakis, G., Fiume, E., and Fournier, A. Tightly-Coupled Multi-Processing for a Global Illumination Algorithm. EUROGRAPHICS '90, Montreux, Switzerland, 1990.

[8] Drucker, S., and Schroder, P. Fast Radiosity Using a Data Parallel Architecture. Third Eurographics Workshop on Rendering, 1992.

[9] Feda, M., and Purgathofer, W. Progressive Refinement Radiosity on a Transputer Network. Second Eurographics Workshop on Rendering, 1991, 139-148.

[10] Feda, M., and Purgathofer, W. Progressive Ray Refinement for Monte Carlo Radiosity. Fourth Eurographics Workshop on Rendering, 1993, 15-25.

[11] Garey, M., and Johnson, D. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York, 1979.

[12] Guattery, S., and Miller, G. On the Performance of Spectral Graph Partitioning Methods. 1995 ACM-SIAM Symposium on Discrete Algorithms (SODA), 1995.

[13] Golub, G., and Van Loan, C. Matrix Computations. John Hopkins University Press, Baltimore, MD, 2nd Edition, 1989.

[14] Goral, C., Torrance, K., Greenberg, D., and Battaile, B. Modeling the Interaction of Light Between Diffuse Surfaces. Computer Graphics (Proc. SIGGRAPH '84), 18, 3, 213-222.

[15] Gortler, S., Schroder, P., Cohen, M., and Hanrahan, P. Wavelet Radiosity. Computer Graphics (Proc. SIGGRAPH '93), 221-230.

[16] Guitton, P., Roman, J., and Subrenat, G. Implementation Results and Analysis of a Parallel Progressive Radiosity. In 1995 Parallel Rendering Symposium, Atlanta, Georgia, October, 1995, 31-37.

[17] Hanrahan, P., and Salzman, D. A Rapid Hierarchical Radiosity Algorithm. Computer Graphics (Proc. SIGGRAPH '91), 25, 4, 197-206.

[18] Naylor, B. Constructing Good Partitioning Trees. Graphics Interface '93. Toronto, CA, May, 1993, 181-191.

[19] Paddon, D., and Chalmers, A. Parallel Processing of the Radiosity Method. Computer-Aided Design, 26, 12, December, 1994, 917-927.

[20] Recker, R., George, D., and Greenberg, D. Acceleration Techniques for Progressive Refinement Radiosity. Computer Graphics (1990 Symposium on Interactive 3D Graphics), 24, 2, 59-66.

[21] Rushmeier, H., Patterson, C., and Veerasamy, A. Geometric Simplification for Indirect Illumination Calculations. Graphics Interface '93, May, 1993, 227-236.

[22] Sillion, F. A Unified Hierarchical Algorithm for Global Illumination with Scattering Volumes and Object Clusters. IEEE Transactions on Visualization and Computer Graphics, I, 3, September, 1995.

[23] Singh, J.P., Gupta, A. and Levoy, M. Parallel Visualization Algorithms: Performance and Architectural Implications. IEEE Computer, 27, 7 (July 1994), 45-55.

[24] Smits, B., Arvo, J., and Greenberg, D. A Clustering Algorithm for Radiosity in Complex Environments. Computer Graphics (Proc. SIGGRAPH '94), 435-442.

[25] Teller, S., Visibility Computations in Densely Occluded Polyhedral Environments. Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1992. Also available as UC Berkeley technical report UCB/CSD-92-708.

[26] Teller, S., and Hanrahan, P. Global Visibility Algorithms for Illumination Computations. Computer Graphics (Proc. SIGGRAPH '93), 239-246.

[27] Teller, S., Fowler, C., Funkhouser, T., and Hanrahan, P. Partitioning and Ordering Large Radiosity Computations. Computer Graphics (Proc. SIGGRAPH '94), 443-450.

[28] Wallace, J., Elmquist, K., Haines, E. A Ray Tracing Algorithm for Progressive Radiosity. Computer Graphics (Proc. SIGGRAPH '89), 23, 3, 315-324.

[29] Young, D.M. Iterative Solution of Large Linear Systems. Computer Science and Applied Mathematics. Academic Press, New York, 1971.

[30] Zareski, D., Wade, B., Hubbard, P. and Shirley, P. Efficient Parallel Global Illumination using Density Estimation. 1995 Parallel Rendering Symposium. Atlanta, Georgia, October, 1995, 47-54.

[31] Zareski, D. Parallel Decomposition of View-Independent Global Illumination Algorithms. Master's thesis, Cornell University, 1996.