Parallel Progressive Radiosity with Adaptive Meshing

Yizhou Yu, Oscar H. Ibarra, Tao Yang

Department of Computer Science, University of California
Santa Barbara, CA 93106, USA

Abstract. Radiosity plays an important role in computer graphics. This paper presents an efficient parallel algorithm for progressive radiosity with adaptive meshing, which adopts a novel static processor assignment strategy and takes advantages of hierarchical computation structure in this problem to minimize communication and balance dynamic load. Our experiments on a Meiko CS-2 distributed memory machine show that this algorithm has achieved good performance for the tested cases.

1 Introduction

The main application of realistic image synthesis in computer graphics is to create simulated scenes of photo realistic quality. Recently radiosity method becomes increasingly popular for determining global illumination in architectural modeling and virtual reality [5, 12] because it accurately portrays illumination effects such as shadows and interreflections in diffuse environments.

An important application requirement is to provide real-time interaction when a user modifies a high-quality synthesized image in a computer aided design process. At the present time, radiosity can be used for such cases but they are too time-consuming to accomplish real-time interaction. A compromise solution to reduce the radiosity time is to use progressive illumination calculation [3, 13]. The basic idea is to conduct a sequence of refinement iterations to improve the image quality. The intermediate result after a few iterations can be used for displaying and a user can revise the design based on this intermediate solution without waiting for the entire computation to end. A further improvement to increase the image quality of radiosity is to incorporate adaptive subdivision [6, 10].

It is important to further reduce the radiosity time since it still takes minutes to see the obvious improvement from one view to another. There have been several parallel algorithms proposed in [1, 2, 9, 11] but speedups for progressive radiosity with adaptive meshing are still low because the computation associated with patches in a 3D space is unstructured and the cost varies during iterations. Furthermore, adaptive refinement of patches may increase the number of patches on some processors dynamically, consequently it worsens the load balance. Thus, load balancing is the main challenge in achieving high speedups [9]. Recent research [2, 9] improves parallel performance by concurrentizing several iterations on different processors. This is called to exploit control parallelism. In this scheme, synchronization is needed when several iterations modify the same patch object and communication overhead is high on distributed memory machines.

In this paper we propose a parallel progressive radiosity algorithm which exploits data parallelism instead of control parallelism. It identifies the hierarchical computational structure in this problem and uses a novel object ordering with static cyclic mapping to achieve load balance and low communication overhead.

2 Progressive Radiosity with Adaptive Meshing

We begin with an overview of the sequential algorithm. The radiosity method assumes that a 3D space contains a set of surfaces, which are perfectly diffuse, i.e. they reflect light with equal radiance in all directions. To get discrete numerical solutions, each surface is further subdivided into a set of patches and this process is called *meshing*. Radiosity is assumed to be constant on each patch and we need to derive this value in order to generate a realistic image. The patch radiosities satisfy the well-known radiosity equation, valid for each sample wavelength:

$$B_i = E_i + \rho_i \sum_{j=1}^{N} F_{ji} B_j \ . \tag{1}$$

The constants and unknown variables are:

- N is the number of patches after meshing.
- B_i is the radiosity(emittance per unit area) of patch i (radiosity) to be solved in the radiosity process.
- E_i is the given self-emitted radiosity of patch i. If $E_i > 0$, patch i is usually called a light source.
- ρ_i is the given reflectivity of patch i.
- F_{ji} is the constant form-factor which is fraction of light flux leaving patch i that arrives at patch j.

This system of simultaneous equations represents the equilibrium energy interchange via multiple interreflections and emissions in the environment.

2.1 Progressive Radiosity

Directly solving (1) is too costly for large N. Progressive radiosity [3] is a kind of iterative method to solve 1. The algorithm is given in Fig. 1. At each iteration, the patch with maximum unshot light flux is chosen to shoot its energy. All patches will receive some part of this flux and update their radiosity values. After each iteration the accuracy of the radiosity for each patch increases, and the partial result can be displayed after several iterations. The complexity of the form-factor calculation is distributed among these iterations and the partial result can be displayed sooner than the Gauss-Seidel based method. Also the computation associated with dark patches with low shooting energy is delayed because they do not contribute light energy significantly in the 3D space.

2.2 Adaptive Meshing

In the first paragraph of Sect. 2, we have assumed that each patch has a constant radiosity. That is actually not true for each patch with a large area. To achieve a good accuracy, adaptive meshing was integrated with radiosity in [4, 6, 10, 15]. In this approach, each patch derived from the initial subdivision (meshing) might be further refined during the course of the computation if the accuracy

real $F_i[N]$; /* *Columns of form-factors* */
typedef real spectrum[# *of wavelengths*]; /* *RGB tristimuli are usually used.* */
spectrum ΔRad, $\Delta B[N]$;
spectrum $B[N]$; /* *Array of radiosities* */
for all patches **do** /* *Initialization loop* */
  $\Delta B[i] \stackrel{v}{=} E_i$; /* *Delta radiosity= self-emittance* */
  $B[i] \stackrel{v}{=} E_i$;
/* *The following is the iterative resolution process.* */
while no convergence() **do** /* *Emission loop* */
  i=patch-of-max-flux();
  compute form-factors $F_i[j]$; /* *Form-factor loop* */
  for all patches j **do**{ /* *Update loop* */
    $\Delta Rad \stackrel{v}{=} \rho_j \Delta B[i] \stackrel{v}{\times} F_i[j] A_j / A_i$;
    $\Delta B[j] \stackrel{v}{=} \Delta B[j] \stackrel{v}{+} \Delta Rad$;
    $B[j] \stackrel{v}{=} B[j] \stackrel{v}{+} \Delta Rad$;
  } /* *End of update loop* */
  $\Delta B[i] \stackrel{v}{=} 0.0$;
} /* *End of emission loop* */

Fig. 1. The progressive radiosity method where symbols, $\stackrel{v}{=}$, $\stackrel{v}{\times}$ and $\stackrel{v}{+}$, indicate that the operations are applied on vectors.

of each patch radiosity does not meet the standard. There are various criteria for such adaptive meshing. We use the following method which performs well on parametric domain of curved or planar surfaces:

- 1) We first subdivide each surface into a triangular mesh. Only patches produced in this initial meshing are called *superpatches*. Each superpatch may be further adaptively subdivided into smaller patches during iterations using the method described in 2). But the *shooting* patch in progressive radiosity with adaptive meshing is always a *superpatch*.
- 2) In each iteration, for each triangular patch in the current mesh as shown in Fig. 2, we compute the radiosity received from the current shooting patch at four points A, B, C and D which are centers of four smaller triangles obtained by connecting the midpoints of the edges of the larger one.
  Then we obtain the linearly interpolated radiosity at D from the radiosities at A, B and C, and compare it with the previously computed radiosity at D. If the difference is larger than a threshold, subdivide this triangle into smaller ones. Otherwise, test linearity at some sample points on AB, BC and CA. If it is nonlinear for any of them, subdivide this triangular patch. We apply the above process recursively on each newly generated patches.

2.3 Form-factor Calculation

We discuss how each form-factor is computed for the above sequential algorithm. Recall the analytic expression for form-factors:

$$F_{ij} = \frac{1}{\pi A_i} \int_{A_i} \int_{A_j} \frac{cos\theta_i cos\theta_j}{r^2} VIS(P_i, P_j) dA_i dA_j \qquad (2)$$

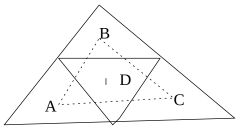


Fig. 2. Subdivision within a triangular patch.

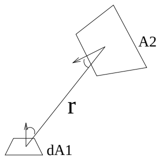


Fig. 3. Form-factor from an area to a differential area.

where the cosine functions and visibility testing, $VIS(P_i, P_j)$, are very expensive to evaluate. That is why form-factor calculation is the main concern in radiosity algorithms.

Given area A_1 in the receiving patch and A_2 in the shooting patch, as shown in Fig. 3, [13] proposed a disk formula for approximating the analytic expression for form-factor from area A_2 to a differential area dA_1 on area A_1. The resulting numerical formula for radiosity at dA_1 due to illumination by A_2 is

$$B_1 = \rho_1 B_2 A_2 \frac{1}{l} \sum_{i=1}^{l} \delta_i \frac{cos\theta_{1i} cos\theta_{2i}}{\pi r_i^2 + A_2/l} \tag{3}$$

where l is the number of sample points on A_2 and δ_i expresses the visibility of each sample point from dA_1.

We adopt this formula because it is desirable to obtain the radiosity at some sample point with a known normal on a curved patch rather than the radiosity received by the whole patch. In our algorithm, the radiosity of each triangular patch is approximated by the radiosity at its center. Thus, in the progressive radiosity algorithm, the first statement inside the innermost for loop should be modified to: ΔRad= the radiosity received by the center of patch j from patch i obtained by using (3).

3 A Parallel Solution

Our approach is to follow the sequential algorithm semantics, exploit irregular data parallelism instead of control parallelism and provide a carefully designed processor assignment strategy. It parallelizes the *for* loop in the algorithm of Fig. 1. The calculation for each patch needs all surface information but does not need knowledge about patches on other processors.

3.1 Initialization and Data Distribution

At the beginning, the albedo and geometrical information of all surfaces making up the objects in a scene are replicated to all processors. After each processor computes the bounding box for each surface, it sets up an octree-based space partition which will be used in the data distribution and form-factor calculation. Then each processor subdivides each surface into a number of large triangular superpatches of approximately the same size and orders all superpatches used for the processor assignment. Notice that the result of this partitioning process is used for the rest of the iterations, and it only takes a couple of seconds. There is no need to parallelize this process.

In the superpatch distribution stage, our algorithm distributes superpatches cyclically on the given p processors based on their ordering numbers. We will discuss the superpatch ordering and distribution strategy in Sect. 4. Now we have two *hierarchies* of data. The upper level contains the octree partition and geometrical information about all surfaces. The lower level contains radiosity and geometrical information on all patches. After data distribution, each processor only owns a subset of superpatches and it keeps a copy of all upper level information to reduce *communication* arising from data dependency in form-factor calculation. Duplicating the data in upper level has little impact on memory requirement. This is because usually we have several hundred of surfaces in a scene but the number of patches after the initial and adaptive meshing is very large.

3.2 Iterative Radiosity Computation

At the beginning of each iteration, each processor picks up the superpatch with the maximum unshot light flux in its own dataset. Then a global MAX reduction operation which costs about $O(\log p)$ is conducted to select the superpatch with the global maximum unshot light flux. Then each processor only works on its own patches to receive radiosity from the shooting patch. It may subdivide a patch into smaller ones by using the adaptive meshing technique discussed in Sect. 2.2. At the end of each iteration, a barrier is called to synchronize the processors and start the next iteration. To increase granularity, we can shoot more than one superpatch at each iteration.

We discuss a *hierarchical* strategy in reducing the computation cost associated with radiosity calculation in a patch subdivision using adaptive meshing. By observation, we find that the lightness of objects in a scene is contributed mainly by a small number of light sources. After they finish shooting, the changes of lightness in this environment are not substantial. On the other hand, the existence of these light sources causes the adaptive subdivision of other patches. But the resulting fine mesh is only useful when these sources possess a large amount of unshot flux. After shooting from bright light sources completes, coarser patch subdivision may be enough because the dark light sources do not lead to significant changes of radiosity values in destination patches. Thus we maintain the hierarchical structure of subdivision. When a shooting patch is selected, we start to compute the impact of this shooting patch on radiosities of a set of patches represented hierarchically in a top-down fashion. We first start from the superpatch and apply the procedure to its child nodes recursively. When the computation at an internal node x shows that the radiosity distribution for subtree nodes rooted at x is smooth enough, the computation can stop at this point.

This hierarchical computing strategy can eliminate the amount of computation substantially because many iterations involve shooting patches with small unshot energies. It also improves the *load balance* because given a set of superpatches after initial meshing, uneven run-time subdivision of each superpatch is the major reason causing load imbalance. The hierarchical computing method eliminates unnecessary computation in a superpatch with a large subdivision tree, and thus smoothes cost difference between different patch subdivision trees. We will show the effectiveness of this hierarchical strategy in our experiments.

3.3 Form-factor Calculation

It is indicated in [2] that the form-factor calculation and the radiosity update require an access to the information regarding *all* patches. That is actually not necessary. One good method for the form-factor calculation is discussed in [13] and reviewed in Sect. 2.3.

We only need the radiosity and geometrical information of the shooting patch, the albedo and geometrical information of the receiving patches and occluding information implied in the octree space partition. Thus each processor only accesses local patch information and the replicated surface data. No interprocessor communication is needed during the form-factor calculation. For the method discussed in Sect. 2.3, visibility test only uses surface geometry. It is done by casting rays from the current receiving patch to some sample points on the shooting patch. This process can be significantly accelerated by using the octree space partition [8]. To accelerate form-factor calculation, we can also fit a coarser mesh for the shooting patch by combining some of the tiny patches. And a larger patch takes the average radiosity of the smaller ones.

4 Processor Assignment for Patches

Our goal is to distribute the amount of dynamic computation evenly among all processors. Since the amount of computation differs for different patches, it is not sufficient to only make each processor have an equal number of patches. The problem of partitioning irregular data space has been studied in other scientific application areas, for example n-body simulations [7]. Usually the quadtree(octree) partitioning method is used and neighboring leafs are directly mapped to the same processor. This mapping may not be effective for our problem, which will be justified in our performance analysis.

Dynamic load balancing that migrates patches between processors looks reasonable. But in practice it suffers high communication overhead on a distributed memory machine. We will use a static assignment for all superpatches to avoid patch migration overhead: each superpatch and its possible child patches generated during adaptive meshing are assigned to the same processor. The *key idea* is to identify superpatches with similar cost involved in adaptive meshing and distribute these superpatches evenly among different processors.

We study the computation requirement associated with each super or non-super patch. At each iteration, most of the computation is spent on the numerical integration for form-factor calculation (Equation (3)), whose complexity depends on many factors, such as the distance between the shooting and receiving patches, the relative orientation of the patches. The following are some facts that we need to exploit in the design of processor assignment to achieve load balance.

- **F1:** Adjacent patches on the same surface have similar orientation because a surface is usually represented by a continuous mathematical function which has continuous normals;
- **F2:** Patches close to each other have approximately the same distance to the shooting patch;
- **F3:** Most subdivision happens at shadow boundaries. If a patch contains a shadow boundary, another patch nearby is also likely to contain a shadow boundary since shadow boundaries are continuous curves. So both these patches are likely to be subdivided into many smaller patches.

Based on these facts, in addition to make processors have the approximately equal number of superpatches, we propose the following strategies for processor mapping.

- **S1:** Adjacent superpatches on the same surface should be distributed to different processors. This will not incur additional communication because adjacent patches have no extra dependence on each other.
- **S2:** Superpatches in different surfaces close to each other should be distributed to different processors based on F2. This heuristic will be applied after S1 because S2 has less constraints.

Since all surfaces are distributed irregularly in a 3D space, there is no obvious ordering between them. To achieve the goal of S1 and S2, we derive a linear order for all superpatches, and then cyclically distribute superpatches to processors based on this order.

4.1 Ordering of Superpatches within the Same Surface

Each surface is assumed to be bivariate and its parametric domain is a square. A triangular superpatch is always subdivided into two smaller ones at its longest edge during the initial meshing. The ordering of superpatches within a surface is defined recursively during initial meshing using the following three rules:

1. Initially a surface is subdivided into four superpatches. Their ordering is clockwise as depicted in Fig. 4(a).
2. If a superpatch is subdivided, the two newly generated superpatches should be inserted into the ordering at the position occupied by the original superpatch. So if superpatch 2 in Fig. 4(a) is subdivided, the ordering is shown in Fig. 4(b).
3. Given the ordering for the current mesh, there exists a route traversing all superpatches following the ordering. As the directed cycle shown in Fig. 4(a), each superpatch has an entry and an exit edge. An entry edge is the one where the route enters the superpatch. An exit edge is the one where the route leaves the superpatch. If a superpatch in the current mesh is subdivided, the relative order of the two new superpatches is defined as follows.
   (a) If the subdivision happens at the entry edge, the new ordering is shown in Fig. 5(a); **(b)** If the subdivision happens at the exit edge, the new ordering is shown in Fig. 5(b); **(c)** If the subdivision happens at the third edge, the new ordering is shown in Fig. 5(c).

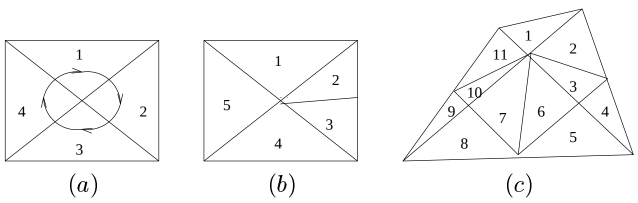
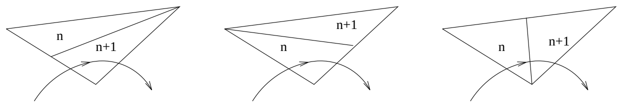


Fig. 4. Ordering after subdivision.

Example:. The ordering for a surface using the above rule is shown in Fig. 4(c). The superpatches on a surface generally have irregular pattern and there is no way to distribute them row-wise or column-wise.



(a) (b) (c)

Fig. 5. Ordering during subdivision.

It can be shown that a continuous route traversing all superpatches following the above ordering always exists after each subdivision. We can see that two consecutive superpatches in the ordering must be adjacent to each other because they share a common edge. This is important because it is most likely that two adjacent superpatches share the same properties, such as there is one shadow boundary on both of them. However, since we are trying to fit a linear ordering onto a 2D or 3D space, we can not guarantee any two adjacent superpatches will be close to each other in the ordering. But this ordering scheme surely can increase this kind of possibility. Another property is that the superpatches within the same larger superpatch are close to each other in the ordering. This means that these superpatches in the same local region, which share the same local properties, are likely to be distributed to different processors.

4.2 Ordering of Superpatches on Different Surfaces

Now we can define the ordering for superpatches on all surfaces using the same octree partition mentioned in Sect. 3.3.

1. Traverse the octree recursively. If a leaf node L_1 is met before L_2, all superpatches on surfaces intersecting L_1 are ordered before those on surfaces intersecting L_2. If a surface intersects more than one leaf, it is only considered the first time it is encountered.
2. If more than one surfaces intersect the same leaf node, they are ordered randomly. But if one surface S_1 is put before S_2, all superpatches on S_1 are also before those on S_2 in the ordering.

4.3 Performance Analysis

It is difficult to provide an analytic result on the effectiveness of load balancing for an arbitrary irregular object distribution in a 3D space and our strategy is a heuristic. We analyze the performance of our algorithm in the following two scenarios to demonstrate the effectiveness of our design strategy. We also compare our approach with the octree-based block mapping. The following parameters will be used in this analysis.

- p is the number of processors.
- T is the number of superpatches for each surface.
- n is the total number of sample points used for all numerical integrations using (3) for a receiving superpatch in one iteration.
- E is the average cost for each sample point when (3) is evaluated.
- γ is the ratio of the number of radiosity updates and patch subdivisions over the total number of sample points for numerical integrations. So $\gamma << 1$.
- C is the maximum amount of time needed for each radiosity update or patch subdivision. It is less expensive than each form-factor calculation.

Scenario 1: Loosely distributed light source-pairs. Suppose there are m pairs of flat object surfaces $\{S_1, S_1', S_2, S_2', \cdots, S_m, S_m'\}$ as shown in Fig. 6. These surfaces are exactly the same except for their spatial positions and orientations. The two surfaces in the same pair are placed face-to-face very close to each other.

Each surface is subdivided into the same number (T) of superpatches. We assume that $p << m$ and if the shooting superpatch and the receiving superpatch are on the same pair but different surfaces, we need at least n_2 sample points for the numerical integrations to achieve the desired accuracy. We also assume that the surface pairs are placed far away from each other so that if the shooting patch and the receiving patch are on different pairs, we need at most n_1 sample points for numerical integrations. Notice that $n_1 << n_2$.

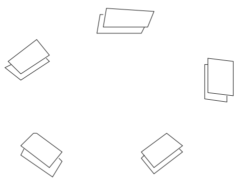


Fig.6. Surface configuration in Scenario 1 when $m = 5$.

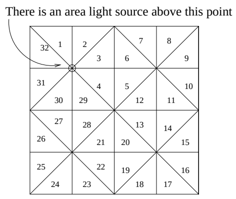


Fig.7. An example of surface configuration for Scenario 2.

For one iteration, let the shooting patch be on S_j. By (3), the sequential time for form-factor calculation in one iteration is $Seq_{form} = En_2T + En_1(2m-2)T$. where the first term is for those superpatches on S_j' and the second term is for those receiving superpatches on other surface pairs.

The sequential time for radiosity updates and patch subdivisions at each iteration is about $Seq_{other} = 2C\gamma n_2 T + 2C\gamma n_1 T(2m-2)$.

When our method is used, we let α be the cost of communication for each iteration. Then the parallel time for each iteration is approximately equal to

$$Par = (E + 2C\gamma)n_2\frac{T}{p} + (E + 2C\gamma)n_1\frac{(2m-2)T}{p} + \alpha\,.$$

So the speedup of K iterations is

$$\frac{Seq_{form} + Seq_{other}}{Par} = \frac{K((E + 2C\gamma)n_2T + (E + 2C\gamma)n_1(2m-2)T)}{K((E + 2C\gamma)n_2\frac{T}{p} + (E + 2C\gamma)n_1\frac{(2m-2)T}{p} + \alpha)} \approx p$$

if the cost of communication (α) is relatively low. Our experiments show that the communication overhead is small.

If we use octree-based block mapping, then superpatches on the same light source pair are assigned to the same processor. Each processor has $\frac{m}{p}$ pairs of surfaces. At each iteration, the processor in which the shooting patch is selected will have the heaviest CPU load which is $(E+2C\gamma)n_2T+(E+2C\gamma)n_1(\frac{2m}{p}-2)T$.

Since $n_1 << n_2$ and $p << m$, the speedup is at most

$$\frac{K((E + 2C\gamma)n_2T + (E + 2C\gamma)n_1(2m-2)T)}{K((E + 2C\gamma)n_2T + (E + 2C\gamma)n_1(\frac{2m}{p}-2)T + \alpha)} \le \frac{n_2T + n_1(2m-2)T}{n_2T + n_1(\frac{2m}{p}-2)T} \le \frac{2}{3}+\frac{2}{3}p\,.$$

Scenario 2: Single light source above a planar surface. The second case has one planar surface. There is a small area light source hanging above the center of the upper left quarter of the planar surface and the position of this light source is very close to that surface. One example is shown is Fig. 7. Based on an analysis [14] similar to that for Scenario 1, we can show our method can reach near optimal speedup(p) while the speedup of octree-based block mapping is much less than p. So our method can balance the load and outperform the octree-based block mapping.

5 Experimental Results

We implemented the parallel radiosity algorithm presented in the previous two sections on a Meiko CS-2 distributed memory machine. Each Meiko node is SUN Sparc Viking with a peak performance 40MFLOP per second and nodes are connected by a fat-tree like fast network with a peak bandwidth 40MB per second. An octree is set up on each processor for data distribution and the acceleration of ray-object intersection.

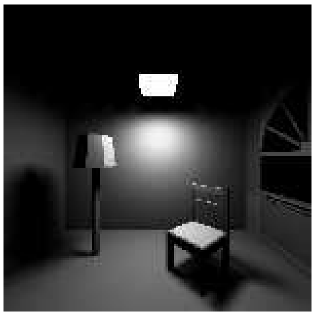


Fig. 8. The left is a room with a lamp, chair and window. The right is a room with tables, stools and some other objects.

We tested our algorithm on three indoor scenes. Two scenes are shown in Fig. 8. The number of surfaces in each scene is between 200 and 300. The number of superpatches after the initial meshing for each scene is around 10,000. 400 iterations were executed for each scene, which is reasonable for interactive design. The sequential time for each scene is about 3 hours.

Scalability. The parallel time speedup for each test scene is shown in Fig. 9. This result is comparable to the recent results for progressive radiosity[2] *without* adaptive meshing, and is much better than previous results for both adaptive and non-adaptive progressive radiosity[9]. Due to the difficulties arising from adaptive meshing, the above experimental result shows that our algorithm deals with irregular and adaptive load variation very well.

Distribution of computation time, idle time and communication overhead. We will show that the load among processors is approximately the same during the radiosity computation. We measure the CPU time used for arithmetic computation for each processor and we call it workload. The workload distribution on 16 processors for 400 iterations in processing the first scene is shown in Table 1. It can be seen that our algorithm has achieved a good load balance. The workload distribution for other scenes is similar.

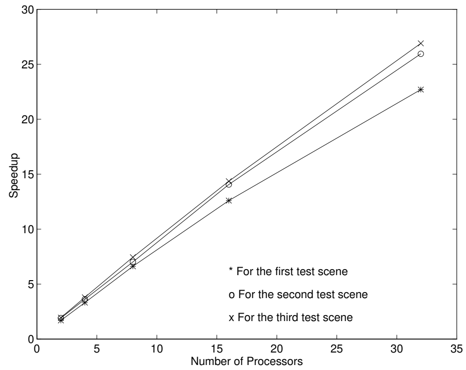


Fig. 9. Speedup for three test scenes.

Table 1. Workload distribution on 16 processors for the first scene.

Proc. ID	1	2	3	4	5	6	7	8
Workload	1314	1308	1303	1332	1313	1509	1286	1334
Proc. ID	9	10	11	12	13	14	15	16
Workload	1577	1291	1303	1259	1572	1301	1659	1314

We have not reached the perfect speedup when $p = 32$ because we lose some performance due to communication overhead and some processor idles caused by small load imbalance. Table 2 lists the average time among all processors spent on communication and the standard deviation of total amount of workload on 32 processors. The communication cost is extremely low, used for selecting the shooting patch and broadcasting the data for the shooting patch. The load imbalance factor is within 10%. That is probably the best we can achieve for a static assignment in responding to the dynamically-changing workload. The previous work [9] using dynamic balancing does not have a good speedup because of high communication overhead.

Table 2. Cost of communication and standard deviation of workload.

	Scene 1	Scene 2	Scene 3
Communication	0.87%	1.21%	1.65%
SDev. of workload	9.94%	7.40%	9.78%

Effectiveness of the hierarchical calculation strategy. We have examined the impact of our hierarchical calculation method which is discussed in Sect. 3.2. In the following, we list the improvement ratio $(\frac{Speedup_{with}}{Speedup_{without}} - 1)$ when p=32. Here $Speedup_{without}$ is the speedup without using this strategy and $Speedup_{with}$ is the speedup using our method. The improvement is around 22% for scene 1, 16% for scene 2, and 24% for scene 3. That is another reason we can obtain a much better performance compared to [9].

6 Conclusions

We presented an efficient parallel progressive radiosity algorithm with adaptive patch refinement using a hierarchical computing and octree-based cyclic mapping scheme. This algorithm can significantly reduce the time in generating high-quality realistic images for real-time interaction. Our parallel algorithm efficiently exploits irregular and adaptive data parallelism using a carefully designed object assignment method. Our experiments show that it has low communication cost and promising scalability.

Acknowledgments This work was supported in part by a grant from the UCSB committee on research, NSF grants CCR89-18409 and CCR-9409695. We thank the anonymous referees for their valuable comments.

References

1. Baum, D., Winget, J.: Real Time Radiosity Through Parallel Processing and Hardware Acceleration. Computer Graphics **24**(1990), No.2, 67–75.
2. Bouatouch, K., Priol, T.: Data Management Scheme for Parallel Radiosity. Computer-Aided Design **26** (1994) 876–882.
3. Cohen, M., Chen, S., et al.: A Progressive Refinement Approach to Fast Radiosity Image Generation. Computer Graphics **22**(1988), No.4, 75–84.
4. Hanrahan, P., Salzman, D., Aupperle, L.: A Rapid Hierarchical Radiosity Algorithm. Computer Graphics **25**(1991), No.4, 197–206.
5. Kawai, J., Painter, J., Cohen, M.: Radioptimization–Goal Based Rendering. Proceedings of SIGGRAPH'93, 147–154.
6. Lischinski, D., Tampieri, F., Greenberg, D.: Discontinuity Meshing for Accurate Radiosity. IEEE CG&A **12**(1992), No.6, 25–39.
7. Liu, P., Bhatt, S.: Experiences with parallel N-body simulation. Proceedings of 6th Annual ACM Symposium on Parallel Algorithms and Architectures(1994) 122-131.
8. MacDonald, J., Booth, K.: Heuristics for ray tracing using space subdivision. the Visual Computer **6** (1990) 153–166.
9. Paddon, D., Chalmers, A.: Parallel processing of the radiosity method. Computer-Aided Design **26** (1994) 917-927.
10. Paulin, M., Jessel, J.: Adaptive mesh generation for progressive radiosity: A ray-tracing based algorithm. Proceedings of Eurographics'94, C421-C432.
11. Recker, R., George, D., Greenberg, D.: Acceleration Techniques for Progressive Refinement Radiosity. Computer Graphics **24**(1990), No.2, 59-66.
12. Schoeneman, C., Dorsey, J., Smits, B., Arvo, J., Greenberg, D.: Painting with Light. Proceedings of SIGGRAPH'93, 143-146.
13. Wallace, J., Elmquist, K., Haines, E.: A Ray Tracing Algorithm for Progressive Radiosity. Computer Graphics **23**(1989), No.3, 315–324.
14. Yu, Y., Ibarra, O., Yang, T.: Parallel Progressive Radiosity with Adaptive Meshing. UCSB Computer Science Technical Report(1996).
15. Yu, Y., Peng, Q.: Multiresolution B-spline Radiosity. Proceedings of Eurographics'95, C285-C298.