Parallel Vertex-To-Vertex Radiosity on a Distributed Shared Memory System*

Adi Bar-Lev  Ayal Itzkovitz  Alon Raviv  Assaf Schuster
Department of Computer Science
Technion - Israel Institute of Technology

Abstract

In this paper we describe the parallel implementation of the Vertex-To-Vertex Radiosity method on a cluster of PC hosts with a Distributed Shared Memory interface (DSM). We first explain how we use stochastic rays to compute the Form-Factor. We then proceed to describe the implementation of this method on top of the MILLIPEDE system, a virtual parallel machine that runs on top of available distributed environments. We discuss a step-by-step process for exploiting MILLIPEDE’s optimization mechanisms. Despite the relatively slow communication medium, the optimization process leads from initial slowdown to high speedups.

*Preliminary results of this work were presented at the Workshop on Algorithm Engineering, Venice, September 1997.
1 Introduction

Fast generation of photorealistic rendering of a scene using computer graphics techniques is one of the most challenging tasks facing researchers today. In the real world, a constant-in-time amount of energy that is emitted by a light source establishes an energy transfer balance between all the objects in the scene in a matter of just a few nano-seconds. It is this state of energy balance we wish to simulate and display as our final stage in the generation of a synthetic scene.

One popular and relatively efficient method for generating realistic scenes is the Ray-Tracing method, which utilizes the reflective and refractive properties of materials to render the scene accurately. In Ray-Tracing, rays are cast from the viewer location towards the virtual scene along paths which reverse those of the original rays in the real-life version of the scene. After striking an object a ray splits into two, the reflection ray and the refraction ray, and the process is then repeated for each of the new rays. The shade which is produced by the initial ray is thus a combination of the shade cast by the object and the shade produced by its two descendant rays.

The Ray-Tracing technique, though effective in scenes containing shiny and highly reflective (specular) objects, will produce unrealistic effects for closed space scenes where most of the effects result from the diffusion of light from the walls and other objects. More on this subject can be found in [5].

The Radiosity method, adapted from the field of Heat Transfer (Sparrow & Cess [13]), attempts to render the scene by simulating this diffusion process. Diffusive materials distribute the unabsorbed energy from light sources in a uniform manner in all directions. The Radiosity method simulates the resulting energy transfer by discretizing the environment, representing it in terms of polygons, and finding an approximation of the energy each polygon receives. The computation eventually assigns to each polygon in the scene a fraction of the total energy which was distributed by the light sources. The energy is then translated into terms of color and intensity, thus providing the shade and lighting of the scene.

Most of the shading which we see in every day life is in fact a result of the diffusive property of substances, which causes the light to appear “soft”, and to shade areas which are not in line of sight of its source, such as the area near the corners of a room or under a table. Thus, although radiosity does not work well for specular materials, it is highly effective in the rendering of such closed scenes, which are composed mostly of objects which diffuse light.

Radiosity also has the advantage of being view independent: once the scene has been generated, a viewer may be placed at any point, and since the energy distribution has already been computed, it will take a short time to generate the appearance of the scene from this specific point of view (POV). Ray-Tracing, in contrast, is view dependent; the full computation must be repeated for every new POV. Radiosity also has the ability to simulate not only lights from light sources, but also the shade caused by heat sources such as furnaces in a plant, thus enabling the simulation
scenes far more realistic than those generated by other methods.

The solution for the Radiosity method is, however, known to be highly exhaustive in terms of both memory and computation, and can be as much as two orders of magnitude slower than the Ray-Tracing technique. In this work we present a parallel implementation for the Radiosity method, using an algorithm based on iterating energy casting. Note that spreading the energy between the elements of the scene typically involves an “all-to-all” type communication which may prevent speedups with naive parallel implementations. In particular, we developed in this work a parallel version of Radiosity on a network of PCs for which the communication media is relatively inefficient (when compared to other parallel computing platforms). Furthermore, we used the millipede system which implements (among its other features, see below) Distributed Shared Memory, sometimes creating implicit bursts of communication due to false or true sharing of variables. In order to obtain speedups in this environment we had to design the data structures carefully, to use optimization techniques that are provided by the millipede system, and to fine-tune the involved parameters.

1.1 The Millipede System

Millipede is a work-frame for adaptive distributed computation on non-dedicated networks of personal computers. [4, 7, 11, 6]. Millipede implements distributed shared memory (DSM) and load sharing mechanisms (MGS); it provides a convenient environment to distributively execute multithreaded shared-memory applications. Millipede provides several techniques for reducing false-sharing and other communication which is created implicitly when using the distributed shared memory. These include a strong support for reduced consistency of the shared-memory pages (thus allowing for duplicate local copies on different hosts), with the consistency type assigned on a per-variable basis, a mechanism for allocating different variables on different pages of memory (thus avoiding ping-pong and false-sharing situations), and instructions to copy-in-and-lock pages of memory (to stabilize the system and optimize resource usage).

A millipede application is issued by a user from the millipede Daemon running on the user machine. The Daemon automatically distributes the application over the network, so that identical instances (copies) of the program are available at hosts that are underloaded. If a node is a symmetric multi-processor then all its internal processors are used transparently by a single instance of an application. Instances of an application share a single virtual space. They communicate in a location-independent way via the DSM mechanism and synchronize using the millipede primitives [6].

Current programming languages available for millipede include parallel versions of C, C++, and the splash macros. These languages are very flexible for multi-threaded applications, use true

---

1Millipede is currently implemented in user-level using the Windows-NT operating system, see also www.cs.technion.ac.il/Labs/Millipede.
shared memory, and are enhanced with multi-consistency declarations for shared variables.

## 1.2 Radiosity - An Overview

Given a geometric representation environment, we can divide it into sets of polygons denoted as patches. The light computation in such an environment involves transmission of energy between the patches until an energy balance is reached.

Based on the theory of energy transfer between objects of diffused material, the relative amount of energy transferred between two patches is dependent on the area which is not occluded by other patches in the scene, and on their geometric relation (angles of planes, distance, size and shape). This geometric relation, called the Form-Factor, represents the relative amount of visibility between a point on a patch and the area of another patch in the scene. For example, if we choose a point in a scene and look towards a polygon, we can estimate the Form-Factor by projecting the polygon on a hemisphere which surrounds the point on its plane. The Form-Factor is computed by dividing the projected area by the area of the hemisphere. For a pair of patches, this definition of the Form-Factor dictates that its value is between zero and one; it is zero when one of the patches in the pair has no point on its surface from which any point at all on the other patch surface is visible. Globally, the definition represents the relative visibility of the whole scene from any given patch. Clearly, in a closed scene, any patch “sees” other patches in all directions, so that the total projected area on the hemisphere is exactly the area of the hemisphere.

After discretizing the scene into polygons, we can view the problem as a set of $N$ equations with $N$ variables. The variables in these equations are the total amount of energy received by each patch from the environment after reaching energy balance, and the coefficients of the equations are the Form-Factors between the given patch and the other patches in the scene.

Prior to the discretization, the differential equation is as follows (in the equations $a \cdot$ denotes a scalar multiplication, $a \bullet$ denotes the inner product with the normal of the plane, and $a \times$ denotes the cross-product of two vectors):

\[
B_i \bullet dA_i = E_i \bullet dA_i + \rho_i \cdot \int_{A_j} (B_j \cdot F_{ji} \bullet dA_j),
\]

where

$dA_i, dA_j$ denote the differential area of points $i$ and $j$ in the direction of the plane, respectively,

$B_i, B_j$ denote the energy gathered in $dA_i$ and $dA_j$, respectively,

$E_i$ denotes the amount of energy emitted from $dA_i$,

$\rho_i$ denotes the reflectivity coefficient of $dA_i$, and

$F_{ji}$ denotes the Form-Factor from $dA_j$ to $dA_i$. 

3
After the discretization of the surfaces into polygons we get:

\[ B_i \cdot A_i = E_i \cdot A_i + \rho_i \cdot \sum_j (B_j \cdot F_{j,i} \cdot A_j) \]  

(2)

The fact that the Form-Factor is essentially a geometric quality of a state between two patches dictates a symmetric relation: \( F_{i,j} \cdot A_i = F_{j,i} \cdot A_j \). Re-organizing and substituting this relation in (2) we get:

\[ B_i = E_i + \rho_i \cdot \sum_j (B_j \cdot F_{i,j}) \]  

(3)

Essentially, the equations simulate the energy transfer from the environment (the patches) towards each patch.

This set of equations can be solved by using Gaussian elimination or the Kramer method for solving sets of equations. The time complexity of the solution is \( O(N^3) \), where \( N \) is the number of patches in the scene. Since \( N \) in most cases varies from a few thousand to a few hundred thousands, clearly the complexity of the above method is too high.

A different approach for solving the set of equations uses the Jacobi method, which approximates the correct solution in iterations. The involved complexity is \( O(N^2) \). This time complexity may be halved by using the Gauss-Seidel technique, which finds an approximated solution by letting each iteration use the approximated solution which was found in the previous iteration.

\( Cohen \ and \ Greenberg[3] \) described a slightly different approach called the Progressive-Refinement Radiosity method. In this approach, for each iteration, each patch in the scene computes the amount of energy that it casts over the rest of the patches in the scene (rather than the amount of energy it receives from each of the other patches). Although it does not alter the solution, this approach enables the algorithm to pick the patch with the largest amount of energy for distribution, thus speeding up the approximation process by focusing on the equations of highest weight in the set. In addition, the iteration process can be stopped at any point and the outcome will still approximate the solution in a uniform way, as the patch with the largest amount of energy has less energy to distribute than a certain threshold.

2 The Radiosity Algorithm

We now present the Radiosity algorithm which is based on the vertex-to-vertex Form-Factor calculation. Our initial experiences with it were reported in [2]. Additional approaches using similar methodology were also introduced by others [9, 1].

2.1 Adaptive Area Subdivision

In this work we refer to a scene as a set of objects comprised of convex polygons (or that can be easily polygonized as such). In most scenes the shape and size of the polygons do not match the
“shading lines” that are created in nature; hence, one needs to decompose a given scene in order to achieve best results. For example, consider a scene of a room with a light-source and a table. The light-source is hanging from the ceiling and the table is in the center of the room, so that the area underneath the table is shaded. This scene will most probably be composed of only one polygon for the floor, which will surely have to be split when the shading is taken into account.

To conclude, a scene should first be split in the best possible way, and only then should the eventual energy of the patches that were created during the process be decided upon. According to the equations, the time complexity increases polynomially with the number of patches, which is determined by the level to which the division is refined. Thus, there is a tradeoff between the time complexity and the accuracy of the result.

Before dividing the scene into new polygons (i.e., new patches), it is important to notice that the major energy contributors according to the equations are the light sources which also define the shadow lines in the scene. Thus, the heuristic choice of which polygons to split takes into account the energy differences between polygon vertices in the original scene. Once the energy difference between two adjacent vertices exceeds a certain threshold, the polygon is split in order to match the shadow borderline.

The technique we chose for splitting the polygons is intuitive, relying on the notion of Polygon Center of Gravity (PCG). When a polygon $P$ is to be divided, we construct its PCG as the “middle point” of its vertices, i.e., the point with the axis values that are the average of the corresponding values of all of the polygon’s vertices. Let $V_{E_{i,j}}$ denote the point in the center of the edge $(V_i, V_j) \in P$. For each $V_i \in P$ with neighbors $j$ and $k$ we construct a new polygon with four vertices: $V_i, V_{E_{i,j}}, V_{E_{i,k}}$, and the PCG of $P$. An example of this division is depicted in Figure 2.1.

![Polygon Center of Gravity](image.png)

Figure 1: Left = initial polygon from the scene. Right = subdividing the polygon into patches using its PCG.

As a result of this process, each divided polygon is split into several new polygons, one for each of its original vertices.
Assume that the energy of each patch can be measured at its vertices. We now denote:

d[A]_i – the differential area around a given vertex \( V_i \). In the discretization process, this area is
represented by the area of the polygons surrounding \( V_i \), and is thus refined at each step of
their subdivision.

\[ \Delta(B_i, B_j) \] – denotes the difference between the energy of two adjacent vertices \( V_i, V_j \).

The recursive division algorithm is as follows:

For each polygon \( P \) in the scene:

1. Construct the PCG of \( P \) and \( dA_i \) for each vertex \( V_i \).
2. Receive energy from each energy source in the scene.
3. For each vertex pair \( V_i, V_j \) such that their energy difference \( \Delta(B_i, B_j) \) is larger
   than a certain threshold subdivide the polygon \( P \) and repeat 1.

### 2.2 Form-Factor Computation

Computing the Form-Factors is the major computational bottleneck of the Radiosity method. In
our approach, all the energy in the scene is divided between the patches, and the energy of each
patch is subdivided between its own vertices. Thus, to calculate Form-Factors, we compute them
between each vertex in the (subdivided) scene and every visible patch. For simplicity’s sake, we
refer to the vertices henceforth as energy sources. This approach helps us deal with simple light-
ources as well, since we can use the light sources in exactly the same way that we use vertices.

Given a vertex \( V_i \), and a patch \( P_j \), we compute the corresponding Form-Factor \( F_{i,j} \) in two stages.
The first stage computes the Initial Form-Factor \( \tilde{F}_{i,j} \) which reflects the relation between \( V_i \) and \( P_j \),
but does not represent any occlusions which might have been caused by other patches in the scene.

Let

\[ \tilde{F}_{i,j} \] denote the initial Form-Factor from \( V_i \) to \( P_j \),

\[ n \] denote the number of vertices in \( P_j \),

\[ R_k \] denote the vector beginning at \( V_i \) and ending at the \( k \)th vertex of \( P_j \),

\[ \beta_k \] denote the angle between the two vectors \( R_k \) and \( R_{k+1} \) which connect \( V_i \) to two adjacent vertices
   of \( P_j \),

\[ N_i \] denote the plane normal of \( P_i \).
Now, $\overline{F_{i,j}}$ can be computed using the following equation for convex polygons.

$$\overline{F_{i,j}} = 1/(2\pi) \cdot \sum_k \{ \beta_k \cdot N_i \cdot (R_k \times R_{k+1}) \}.$$  \hspace{1cm} (4)

Figure 2 depicts the variables used in this equation.

![Diagram of angles and relations in the calculation of the initial Form-Factor between a point in space and a patch.]

The derivation of this formula as well as other formulations for Form-Factor calculation can be found in [3] and [12].

Given an energy-source $V_i$ and a patch $P_j$, we compute $\overline{F_{i,j}}$ by using Equation (4). Next we need to calculate the occlusions for each source of energy. We achieve an approximation of this occlusion by casting stochastic rays. Each ray is cast from a random point in the area near $V_i$ ($dA_i$), to a random point on patch $P_j$. Both the start point and the end point of the ray are selected by using a uniform probability distribution.

If $N_j$ rays are cast towards $P_j$ with Form-Factor $\overline{F_{i,j}}$, $F_{i,j}$ is approximated by calculating the number of hits that actually occurred on $P_j$, i.e., the integral over the non-occluded area of $P_j$ from $V_i$’s POV. The error function in this case can be computed by using the Poisson probability distribution.
Note that $N_j$ is directly related to the energy of the source $V_i$, and to the size of $\widehat{F}_{i,j}$. Let $n$ denote the number of rays cast over the whole hemisphere with an acceptable error function. Let $\tau$ denote the ratio between the energy of $V_i$ and some pre-specified value, i.e., this ratio limits the number of rays cast for a single energy source as a function of its energy. Now, the number of rays $N_j$ to be cast over $P_j$ is calculated as follows:

$$N_j = \tau \cdot \widehat{F}_{i,j} \cdot n. \quad (5)$$

To summarize, the Form-Factor for $V_i$ and $P_j$ is constructed as follows:

- Compute $\widehat{F}_{i,j}$.
- Use (5) to calculate $N$.
- Let $N_{hit}$ denote the number of ray hits, and initialize $N_{hit} \leftarrow 0$.
- For each ray cast:
  - Randomly pick a point on the surface $dA_i$ in the area near $V_i$.
  - Randomly pick a point on $P_j$.
  - Cast the ray, and if a hit occurs increase $N_{hit}$ by 1.
- Compute $F_{i,j} = \widehat{F}_{i,j} \cdot \frac{N_{hit}}{N}$.

Notice that in order to calculate the Form-Factor for a point light-source, the total energy of the source should be halved because the computation refers to a full sphere rather than to a hemisphere. We treat a light-source as two separate hemispheres, split the energy equally between them, then use the equations as shown above. The only exception to this rule is the origin of each ray, which is located at the light-source, and not in the nearby neighborhood as in the case of a vertex.

### 2.3 The Progressive Refinement algorithm

We now describe the global algorithm which uses the *Progressive Refinement* approach. Let

- $F_{i,l}$ denote the Form-Factor between a given light-source $l$ and $dA_i$,
- $B_{i,l}$ denote the energy transferred from a light-source $l$ to $V_i$ (which depends on $F_{i,l}$),
- $\Delta B_i$ denote the energy that $V_i$ received since the last time it casted energy, and
- $B_i$ denote the total energy $V_i$ received from the beginning of the process.
The algorithm is as follows:

1. Compute the Form-Factors from each light-source over all polygons in the scene. Cast energy at the polygons accordingly.
2. Subdivide the scene into patches and repeat step 1 until a good division is obtained.
3. Having the final division, cast the energy of the light-sources towards the patches in the scene.
4. \( \forall V_i \) assign: \( B_i = B_{i;1} \), \( \Delta B_i = B_{i;2} \).
5. Repeat the following operations: For each vertex \( V_i \) whose \( \Delta B_i \) is larger than the threshold:
   a. for every patch in the scene \( P_j \) compute \( F_{i,j} \) (first iteration only).
   b. Distribute \( \Delta B_i \) between all patches; \( B_i \leftarrow B_i + \Delta B_i \).
   c. \( \Delta B_i \leftarrow 0 \).
   d. Insert \( V_i \) at the end of the vertex list.

The algorithm uses two different notations for the energy. The first is the global energy \( B_i \) which the patch does not absorb, and which gives the patch its shade when the scene is rendered. The second is the energy received by \( V_i \) from other vertices in the scene during their energy distribution phase. The latter is added to the energy gathered by \( V_i \) (denoted \( B_i \)), and is distributed as \( \Delta B_i \) to all the patches in the scene when \( V_i \) reaches the top of the list. In the described algorithm, each vertex is inserted into a sorted list of vertices, where the sorting is done by the \( \Delta B \) of each vertex. A vertex is picked for energy casting from the head of the list, i.e., the most significant energy contributor at that current time.

3 Parallelizing The Algorithm

The Radiosity algorithm which was initially designed as a sequential application, was modified to work efficiently in parallel. This was done in two main steps. The first step was to analyze the potential maximal level of parallelism of the problem, i.e., to locate places in the code where work could potentially be carried in parallel. We put in minimal code modifications to make the application run concurrently on the millipede system and verified the correctness of the outcome. The second step was to improve the performance by handling three factors: true sharing, false sharing and the scheduling policy. The results show that the second step was vital for improving the performance and obtaining speedups in a loosely-coupled distributed shared memory environment.

3.1 The Naive Parallelization

The Radiosity algorithm is based on iterating energy casting as described earlier, where each step is in fact independent of the other steps. Parallelism can be found mainly in three places. The first
is the subdivision stage where the scene is split among the hosts so that each of them computes the subdivision on the polygons it receives. The second is the main stage of the algorithm where each vertex is assigned to a different host, which calculates the Form-Factor and distributes the energy difference to the other vertices. The last stage is the real-time display of the scene.

Parallelizing the subdivision stage is straightforward and we proceed to describe the second and third stages. The implementation of the second stage maintains for each vertex \( V_i \), a data structure which contains the following information: its current energy \( B_i \); its current delta energy \( \Delta B_i \) (which the vertex can further distribute to the other vertices); and the new delta energy \( \hat{\Delta}B_i \) (received by \( V_i \) when some other vertex \( V_k \) distributes its \( \Delta B_k \) towards \( V_i \)). Calculating the Form-Factor and distributing the energy can therefore be done in parallel by using following algorithm:

**Parallel Step:** for vertex \( V_i \) with non-zero \( \hat{\Delta}B_i \) do:
1. Lock \( V_i \) and \( \hat{\Delta}B_i \).
2. \( \Delta B_i \leftarrow \hat{\Delta}B_i, B_i \leftarrow B_i + \hat{\Delta}B_i \).
3. Unlock \( V_i \) and \( \hat{\Delta}B_i \).
4. If the array \( F_{i,j} \) does not exist then compute it.
5. Distribute the \( \Delta B_i \) according to the Form-Factor array.

After initialization, when some initial energy is distributed to the vertices, the algorithm spawns **millipede** jobs which execute the Parallel Steps concurrently for all vertices.

Mutual exclusion should be enforced in two places. The first appears when a vertex \( V_i \) takes the \( \hat{\Delta}B_i \) and adds it to its \( B_i \) and its current \( \Delta B_i \) before distributing its energy to the scene (Parallel Step, stages 1 through 3). No concurrent update of the \( \hat{\Delta}B_i \) is allowed at that time. The second place is when the \( \hat{\Delta}B_k \)'s of the other vertices is modified, an operation that should be taken exclusively by a single vertex at a time (Parallel Step, stage 5).

**Using Millipede**

**Millipede** is designed to support multithreaded parallel programming on top of clusters of PCs. Therefore, the code changes for parallelizing the sequential Radiosity algorithm were minimal. Basically each vertex could execute the Parallel Step independently of all other vertices, thus making it a parallel execution task (a job in **Millipede** terminology). Memory allocation and deallocation operators **new** and **delete** were overloaded to use **Millipede** interface API for allocating memory from the shared memory. **Millipede** atomic operation **faa** (Fetch And Add) was used to preserve mutual exclusion when the new delta energy of the current job vertex was updated, or when this value was updated for the other job vertices.

Starting a new job for each vertex turned out to cause thrashing and calculation of redundant data. Thus we created execution jobs for a limited number of vertices that were picked from the
beginning of the vertex array after the sorting phase, i.e., only those vertices with the highest energy to be cast are handled exclusively by a dedicated job.

Scheduling is done by a special job, the job manager, which is in charge of scheduling new jobs, as follows:

while(not finished)
  1. Sort the vertices according to their energy.
  2. If the number of working threads is below a minimal number, then for each vertex whose energy is above the threshold spawn a new job running the Parallel Step.
  3. Sleep $\Delta$ milliseconds (let the working threads cast some more energy before the next round).
end while

In Figure 3 we see that the naive implementation performs poorly. When the number of machines grows, the computation slows down and the parallel algorithm has no advantage over the sequential one.

![Graphs showing performance improvement](image)

Figure 3: Two versions of the parallel Radiosity algorithm. In the left graph we see that the naive implementation performs poorly when the number of machines involved in the computation increases. In the right graph we see that performance can improve by more than 450 percent over performance of the naive for the fully optimized implementation, when the number of machines grows to 5.

### 3.2 The Optimizations

The parallel implementation, as described above, suffers from several inefficiencies which greatly influence the performance. We attempted to increase performance by focusing on three factors: the
sharing of the data structures, the false sharing in the algorithm implementation, and the scheduling policy. We show that each change greatly improves the execution time, while the combination of the three results in good speedups for various scenes, as can be seen in Figure 4.

![Graphs showing speedups and execution times for scenes](image)

Figure 4: Speedups for three scenes, two relatively small ones (numbers 1 and 2, each less than 200 seconds sequentially), and one relatively large (number 3, about 2000 seconds). For problems with high computational demands (like scene 3) good speedups are achieved.

### 3.2.1 True Sharing

Calculating the Form-Factor of a vertex is a time consuming operation. The calculation involves reading the characteristics of other vertices, performing the mathematical operations and finally storing the Form-Factor in the private data structure of each vertex. In terms of DSM operations, calculating the Form-Factor involves *reading* from other vertices data structures and *writing* to that of its own vertex.

On the other hand, distributing the vertex energy (called *shooting Radiosity*) is a very short operation, and requires merely updating the data structures of other vertices according to current values of the vertex’s own data structure. In terms of DSM operations, distributing the energy involves *reading* from the vertex’s own data structure and *writing* to the data structures of other vertices.

In the naive parallel algorithm a job does both operations, namely, calculating the Form-Factor (if required) and distributing the energy to other vertices. Thus, in the naive parallelization, a true sharing of the vertices data structures occurs. A data structure of a vertex $V_i$ is shared as follows:

- By each job which calculates the Form-Factor of any vertex $u$, to which $V_i$ is visible (shared for reading).
• By each job that distributes the energy of its vertex \( u \) to the visible vertex \( V_i \) (shared for writing).

• By the job to which vertex \( V_i \) is assigned for calculating the Form-Factor (shared for reading), or for distributing its energy to other visible vertices (shared for writing).

The sharing of the data structures by several jobs leads to massive page shuttling and long delays in execution. Note, however, that the Form-Factor calculation takes the majority of the computation time and has a high computation-to-communication ratio (vertices’ data structures are only being read). Therefore, in our solution, only Form-Factor calculations are carried out in parallel. All energy distribution is done on one of the machines, while calculations of the Form-Factors are sent to other machines to be computed concurrently. As a result of this change, access to shared data is reduced significantly. The data structures of the vertices are accessed when they are required by a remote job for reading (i.e., the job is calculating the Form-Factor of that vertex), or by local jobs for writing (i.e., they are distributing their energy to this vertex).

The Parallel Step therefore changes. New jobs are spawned, executing Parallel Step 1 (see below) in the case that the Form-Factor has not yet been calculated, and Parallel Step 2 (see below) otherwise.

**Parallel Step 1** for a vertex \( V_i \): calculate its Form-Factors towards all patches.

**Parallel Step 2** for vertex \( V_i \) with non-zero new delta energy:

1. Copy the \( \Delta B_i \) to the current \( B_i \) and the current \( \Delta B_i \).

2. Distribute the current \( \Delta B_i \) of \( V_i \) according to the Form-Factor to the \( \Delta B_k \)'s of the vertices that are visible to/from \( V_i \).

Figure 5 shows the performance boost of the improved algorithm when the parallel steps are separated into local and remote execution. The improvement is the result of a dramatic reduction in the sharing of the vertices’ data structures. In fact, for the sake of locality of memory reference, the potential level of parallelism is reduced such that distributing the energy (shooting Radiosity) is not carried out concurrently.

### 3.2.2 False Sharing

A known phenomena in page based shared memory systems is false sharing, in which two or more distinct variables that are used by distinct machines are placed on the same memory page. Thus, although there is no true sharing, the page will frequently move between the machines as if its data was shared.

It turned out that the naive parallel Radiosity implementation contained several falsely shared variables in the vertex structure, which led to significant performance degradation. Among others,
Figure 5: Maximizing the level of parallelism does not give optimal execution times. When the creation of remote concurrent jobs is restricted to those calculating the Form-Factor, then an improvement of 60-200 percent is achieved due to less sharing and increased locality of memory references.

This structure contains the values that reflect the energy in the node \((B, \Delta B \text{ and } \Delta\hat{B})\). This data is referenced only at the stage of energy distribution, and is not required during the Form-Factor calculation. Thus, the false sharing on the vertices’ data structure can be avoided by extracting the energy fields to a stand-alone allocation. As a result, jobs which calculate the Form-Factor do not (falsely) share data with shooting Radiosity operations, and thus would not be disturbed by energy distribution carried out concurrently by other vertices.

Figure 6 shows that a performance improvement of 30-50 percent is obtained when the energy variables are excluded from the vertex data structure to a stand-alone allocation. There were several additional falsely shared variables in the naive implementation, and we used the same technique for them.

An additional optimization method we used was padding the allocations to an integral number of pages. This operation also solved many of the false sharing situations.

### 3.2.3 Scheduling Policies

The previously described naive scheduler wakes up once in every \(\Delta\) milliseconds. It sorts the array of vertices according to their energy level, and then spawns new jobs for handling the most significant vertices in the scene at this point. It turned out that a fine tuning of the value of \(\Delta\) is very important. When \(\Delta\) is too small, too many jobs may end up working concurrently on different vertices, making redundant calculations which may be insignificant in terms of energy dispersal. Also, choosing too small a value for \(\Delta\) causes frequent reads of energy information by the scheduling job (for its sorting task), which may cause data races with those jobs updating the energy values.
Figure 6: False sharing causes a slowdown of 30-50 percent in the potential optimal execution time of the parallel Radiosity algorithm. The graph shows the effect of solving it by separating the related allocations. The right graph shows some performance degradation on a single machine, which is the result of an additional level of indirection in accessing some variables. In a distributed system this overhead can be neglected.

in the vertices. At the other extreme, choosing too large a value for $\Delta$ might cause idle machines to wait too long for the scheduler to wake up and spawn new jobs.

Figure 7 shows how performance is dependent on $\Delta$. Further optimization may adjust the value of $\Delta$ dynamically during execution. This requires further study.

We also tried another approach in which the scheduler is set to be event triggered. We used the integrated mechanism for message passing in millipede (named mjec) to implement the scheduler in the following way.

When the Parallel Step ends, the job sends a FINISHED message to the scheduler, which triggers the scheduler to spawn new tasks.

Upon a FINISHED message:

1. Sort the array of vertices according to their energy.
2. For each of the first several highest-energy vertices spawn a new job running the Parallel Step.

Figure 8 shows that the event triggered scheduling policy performs better than the eager $\Delta$ scheduling even when using the optimal $\Delta$ value for the naive scheduler.
Figure 7: The scheduler sleeps for $\Delta$ milliseconds between successive iterations. The optimal $\Delta$ for the scene benchmark turns out to be approximately 100 milliseconds. Execution time using five machines is very high when $\Delta$ values are much larger than the optimal one.

Figure 8: The naive scheduler performs 15-20 percent worse than the event triggered scheduler. Results in the graph were taken with the optimal $\Delta$ values for the naive scheduler.
Conclusions

In this work we implemented a parallel Radiosity rendering algorithm on a loosely-coupled cluster of PCs. The initial naive algorithm did not perform well, and in fact the computation time increased with the number of machines. We then applied a handful of tunings and optimization techniques which eventually made the algorithm execute efficiently, achieving speedups close to optimal.

Previous related works such as [8] use a different approach in which the given scene is divided into sub-scenes, each assigned to a different processor. The communication pattern thus reflects the energy casting from one sub-scene to another, and seems suitable for implementation in a message passing parallel system. As described throughout the paper, our approach seems suitable for parallelization over a distributed shared memory environment such as millipede.

This project was carried on the millipede system which provides an easy to use environment for parallel programming on top of distributed clusters of PCs. In fact, the millipede concept was designed to promote this precise modus operandi (in the process of parallel programming): a first easy phase of porting an existing sequential code to the parallel domain, and then, in a follow-up phase, a step-by-step gradual optimization towards efficiency and improved speedups.

References


