EPCC-SS96-05

Light, The Universe and Parallel Radiosity

Alex Brodsky

Abstract

Radiosity provides an effective method for modelling a diffuse environment. Unfortunately this method is computationally expensive. The main cost of the radiosity method is the calculation of form factors which takes approximately 95% of the computation time. Through the use of a task farm the computation of form factors can be parallelised, on a Massively Parallel Processor (MPP) super computer like the CRAY-T3D. This creates a relatively good linear speed up.

A different approach is to use a Symmetric Multi-Processing (SMP) model. Threads, instead of processes, are used to perform the work using a task farm. Apart from the thread's stack space all memory is global. Communication and synchronisation takes place using semaphores and conditions instead of message passing. This model reduces the amount of communication and synchronisation required between the master and the slaves. The size of the environment is limited only by the total amount of virtual memory available to the process.

Both models provide effective linear speed up solutions to a computationally expensive problem. The task that follows is to improve the algorithm itself and possibly reduce the prohibitively expensive $O(n^2)$ scaling of the problem.

1 Introduction

The holy grail of computer graphics is to generate a complete realistic image of either a real or an imaginary environment. While the tools to capture and propagate realistic images have existed for decades, the ability to generate realistic images of a specified environment still eludes us. The most popular technique for generating spectacular if not completely realistic pictures is ray tracing. This technique works extremely well with highly specular and transparent surfaces. However the majority of environments consist of mostly diffuse surfaces like walls, ceilings, tables, floors and boxes. Ray tracing does not handle diffused surfaces, but radiosity a technique developed about a decade ago does.

Ray tracing operates on the principle of tracing each beam of light from the view point through its many reflections, to a light source. However a diffuse surface does not reflect a beam of light but scatters it in an evenly distributed pattern. Tracing each beam of light from this distribution is computationally impractical due to the number of beams reflected from a diffuse surface. Radiosity on the other hand works on the principles of energy emission and conservation, and is specifically designed for diffuse environments.

Unfortunately radiosity is a very computationally intensive. The general algorithm scales as $O(n^2)$ with the size of the environment and requires excessive use of floating point operations. An average environment of 10^4 patches requires hours of processing time on a high end work station to complete a single image. Once the radiosity completes, the image can be rendered very quickly from any point of view, but the main step of determining the radiosities of all patches is quite costly. This is probably the main reason why radiosity has not become a popular technique for creating images, its heavy computation and memory requirements are just to great to be timely used. But all is not lost.

One approach to speeding up the algorithm is to parallelise it. In our investigation of different parallelisation methods we looked at two different models, the Massively Parallel Processor (MPP) model and the Symmetric Multi-Processor (SMP) model. In the MPP model the number of processors is on the order of 10^2 while the communication between the processors is considered relatively expensive. The SMP model takes the opposite approach. There are usually no more than 16 processors per SMP machine, but all processors share the same global memory. Thus communication is relatively cheap but the number of processors is significantly reduced. During our investigation we created two implementations, the first was implemented using the MPP model, and was targeted at reducing the computation time of the algorithm. The second was implemented using the SMP model and was also implemented to maximise the speed of the algorithm. In this paper we propose but do not implement a third alternative which when implemented in the MPP model would trade off some of the speed for the ability to handle extremely large environments on the order of 10^6 patches. The computation time required for this algorithm is substantial.

All implementations use the same basic radiosity algorithm known as progressive refinement. The run time of the algorithm depends on the amount of accuracy required in the image. The more iterations that the algorithm runs, the greater the convergence to the full matrix solution and hence the accuracy is greater. In almost all the cases this algorithm is significantly more efficient than the full matrix solution and significantly faster.

This paper first describes the theory behind radiosity and general description of the technique. Both the full matrix and the progressive refinement variants are covered. The discussion then proceeds to discuss possible parallelisation strategies, which were either implemented, deferred or discarded. The main sections deal with the details of the two implementations. Afterwards we discuss the performance of the two algorithms. We end of with the conclusion and some words about future work. One clearly visible piece of work is the implementation of the proposed solution, another is to go back to the drawing board and try to come up with a possibly faster, better algorithm.

To produce an image there are three basic steps. The environment must first be created, this is usually done by hand. The radiosity algorithm is then applied to the environment. Once completed each patch in the environment has an associated set of values which describe the amount of light that radiates from the patch. The final step is to render the image. from a certain view point. Since the radiosity step is view independent, and the rendering step is relatively cheap, walk-throughs of the environment can easily be generated once the radiosity step is completed. In this paper we are only concerned with the radiosity step, which is the bottleneck of the entire mechanical part of the process.

2 Principles of Radiosity

Radiosity has its roots in heat transfer models to account for the transfer of heat between elements of the environment[8]. A similar approach developed by Goral et al at Cornell University in 1984[9], is used to create some of the most realistic looking scenes to date.

2.1 Radiosity Theory

Radiosity works on the assumption that the total amount of light in the system must be conserved. Therefore the amount of light that a surface reflects can be no more than the amount of light that another surface receives. Radiosity is basically the amount of energy which radiates from the surface. We define radiosity, B, as the amount of energy per unit area emanating from the surface per unit time. This is the sum of the emitted and reflected energy[9].

$$B_i dA_i = E_i dA_i + R_i \int_j B_j F_{ji} dA_j$$

Where E_i is the energy being emitted by the surface. The integral multiplied by the reflection coefficient R_i represents the total energy received from all other surfaces in the environment. The F_{ji} is the form factor and denotes the fraction of energy from surface j which falls on surface i.

To perform this calculation for every point in a continuous environment is out of the question, therefore the environment is divided into small elements called patches. Each patch is a polygon consisting of 3 or 4 vertices, a non zero area dA_j and is treated as a surface with isotropic radiosity. The integral then becomes a summation of the form:

$$B_i A_i = E_i A_i + R_i \sum_{j=1}^{n} B_j F_{ji} A_j$$

This can be simplified by noting the relationship between patch i and j which is that $F_{ji}A_j = F_{ij}A_i$[8]. Thus the above equation reduces down to:

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

This equation for patch i is one of the n equation in an environment of n patches. A system of these linear equations must be solved in order to determine the amount of light being emitted by each patch in the environment. The values for E_i are non zero if patch i is part of a light source, otherwise the emission is zero.

2.2 Form Factors

In order to solve the afore mentioned system we first need to find the form factors for each patch. The form factor represents the fraction of energy emitted from patch j which lands on patch i. The form factor is defined by the equation[6]:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j dA_i$$

Where ϕ_i is the angle between the normal of patch i and the vector from dA_i to dA_j. Similarly ϕ_j is the angle between the normal of patch j and the vector from dA_j to dA_i. Except for some trivial cases, the above integral cannot be solved analytically. Solving the integral numerically for each of the form factors is prohibitively expensive especially when there are $O(n^2)$ form factors with the average size of n being on the order of 10^4.

To approximate this integral we assume that the patches are relatively large distances apart and that patch i is isotropic. We can approximate the form factor using the differential to finite relation[3]:

$$F_{dA_iA_j} = \int_{A_j} \cos\phi_i \cos\phi_j dA_j$$

According to the Nusseult analogue[8], we can project patch j onto a hemisphere centred on dA_i and consider this equivalent to projecting the patch on surface A_i[9]. By projecting patch j onto the hemisphere and determining the area of the projection, we can easily compute the form factor F_{ij}. However the last part is itself non-trivial. In order to simplify the last operation the hemisphere is replaced with a hemicube[3]. Given the Nusseult analogue the hemicube will give a good approximation of the hemisphere.

To determine the surface area of the projected patch on the hemisphere, we would have to break up the hemisphere into a small solid angles and then sum the solid angles onto which the patch was projected. Each small angle or delta angle represents part of the form factor and is known as a delta form factor[3]. It is significantly easier to divide the hemicube into delta patches than a hemisphere. Each of the 5 faces is divided into square pixels. For each pixel the delta form factor can be calculated once and stored in a look up table. When a patch is then projected onto the hemicube, the delta form factors for all covered pixel are summed together giving a good approximation for the form factor. The delta form factors can be calculated using two equations. The first equation is used for the main face of the cube, the face which is parallel to the patch. The second equation is used for the other four side faces which are all perpendicular to the patch. The two equations for the top and side faces are respectively:

$$\Delta F_q = \frac{1}{\pi(x^2 + y^2 + 1)^2}\Delta A$$

$$\Delta F_q = \frac{z}{\pi(z^2 + y^2 + 1)^2}\Delta A$$

Due to symmetry of the hemicube only one eighth of the delta form factors need to be stored. This provides a sizeable savings in the overhead cost of the system where an average hemicube has around one million pixels per face.

In order to efficiently compute form factors we can project all patches sequentially onto the hemicube and then sum up the delta form factors in one pass of the entire hemicube. In order to achieve this, the pixel buffer of each face of the hemicube becomes a Z-buffer. Thus if more than one patch projects onto the same pixel, the nearest of the them will be recorded as being projected onto that pixel. This also allows us to deal with occluded patches. All form factors for patch i can be calculated in one pass. To calculate form factors for all n patches requires n passes.

The calculation of form factors takes $O(n^2)$ time and requires memory of $O(n^2)$. Once the form factors are calculated all that is left to do is to solve the system of linear equations.

2.3 Full Matrix Solution

The final step is to solve the system of linear equations of the form:

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

This can be rewritten in the standard form of:

$$A\vec{x} = \vec{b}$$

Where A is the matrix of reflection coefficients, $\vec{x}$ is the vector of patch radiosities and $\vec{b}$ is the vector of the patches' initial emissions. As mentioned previously element i of the $\vec{x}$ vector is non zero if patch i is a light source.

Depending on the target architecture there are many different methods for solving the full matrix, the most common one mentioned in the literature is the Gauss-Siedel iterative method[3]. However this method also takes $O(n^2)$ time and requires $O(n^2)$ space. The result is that he amount of space required for an environment with 10^4 patches is approximately 10^9 bytes and a computation time of days. In order to avoid such demand on resources an iterative radiosity solution was developed by Cohen et al. known as progressive refinement.

2.4 Iterative Refinement Radiosity

The full matrix solution is very computationally expensive, which perhaps explains why radiosity has not been very popular in the graphics community to date[9]. In 1988 Cohen et al. developed an iterative version of the radiosity algorithm known as progressive refinement. The idea is to re-arrange the basic algorithm so that it is not necessary to store the entire matrix of form factors. Instead we generate the form factors as they are needed, use them and then throw them away.

The entire re-arrangement can easily be done by changing the perspective of the algorithm. The full matrix solution gathers the light for each patch from all other patches and stores it. Instead of gathering the light from all patches to each patch, we can shoot the light of patch i to all other patches. Since light is conserved, the amount of possible light that patch i can send, once it has been shot, is zero, at least until other patches are shot and patch i receives some more light. At each iteration the patch i with the most unsent light is shot to all other patches[2]. Therefore each patch $j \neq i$ receives:

$$\Delta B_j = \rho_j U_i F_{ij} \frac{A_i}{A_j}$$

Where ΔB_j is the amount of extra light that patch j will radiate after this iteration, ρ_j is the reflection coefficient of patch j, U_i is the amount of unsent light radiating from patch i, F_{ij}, A_i, A_j are the form factor and areas of the respective patches. The ΔB_j is then added to the permanent radiosity value and to the unsent radiosity of patch j which will be used in the next iteration. The largest unsent value is kept in a temporary variable to be used in the next iteration. After the current iteration completes the U_i is zeroed since all light from patch i has been shot[2]. Since the patches are shot in an implicit sorted order, radiosities converge relatively quickly because as much light as possible is shot during each iteration. On average the form factors $\{F_{ij} : j = 1..n\}$ for patch i will be use only once. Thus when patch i is shot the form factors can computed, used for the shooting process, and then thrown away. This allows the algorithm to use linear memory and is much faster than the full matrix algorithm because the progressive refinement algorithm scales as $O(mn)$ where m is the number of iterations and n is the number of patches. Since m is usually less than n and usually grows at a slower linear rate, the overall scaling of the algorithm is significantly better. The m value is usually picked by the user and is a measurement of the amount of detail desired. A greater number of iterations imply a greater amount of detail, although the amount of detail gained per iteration decreases as the number of iterations increase. This makes the the progressive refinement algorithm quite good for generating the image stages and letting the user indicate when sufficient detail has been reached.

2.5 Rendering

Once the radiosity algorithm has completed, each patch in the environment has the appropriate colour intensity associated with it. In order to view the image we select a view point and render then image projecting all polygons in the environment onto a two dimension viewing screen. The viewing screen is itself a Z-buffer so the final image in the buffer is what the environment would look like if we were to view it from that point. To make the image more realistic different shading and colour blending techniques can be used to smooth out the colour differences between adjoining patches. This is the post-processing phase and is not significant to the radiosity algorithm. It is however a required step if the image is to be viewed. This step can be repeated multiple times from different view points without having to rerun the radiosity algorithm. If the environment is changed in any way the radiosity algorithm will have to be run again.

3 Radiosity Parallelisation Methods

During the initial stages of this project, several methods for parallelising the radiosity system were discussed, accepted or rejected based on a set of criteria. The most important criteria was the resulting speedup factor that would be achieved. The next criteria was the the ability to handle large environments, in the neighbourhood of 10^6 patches. The third criteria was simplicity of implementation. The fourth criteria which overlapped with the previous three was the amount of communication involved for each of the schemes. Although this criteria could be treated as part of any of the other three criteria, its strong correlation between all three other criteria made it a separate criteria in this case. Based on these criteria six different parallelisation approaches were considered.

The first scheme proposed to parallelise the generation of the hemicube. In this approach each processor element (PE) would be assigned a face of the hemicube to compute and sum. The main face of the hemicube would actually be treated as two faces so that all faces would be of equal size. The six summations would then be reduced to the main PE and the form factors utilised in the shooting of the current patch. This scheme would be relatively simple to implement, and the amount of communication required would be minimal. The initial distribution of the environment to all PEs does not pose any great communication overheads and the reduction of the form factors is the minimal communication that occurs in any of the schemes. The problem with this scheme is that it does not scale well on more than six PEs. Furthermore the environment size is restricted to the amount of memory found on one PE. Therefore the system would not be able to handle large environments on the order of 10^6 patches. Based on these criteria the scheme was rejected.

The next scheme to be considered involved a task farm configuration. In this scheme the master PE would be responsible for shooting the patches, but the job of calculating the form factors would fall on the slaves. Each slave would have a replicated environment and would only produce the form factors which were suggested or requested by the master. While this scheme is not as simple as the previous one, especially when the problem of synchronisation arises, the over all complexity is manageable. The important factor is that it will easily scale to any number of PEs, and as the problem grows bigger the scaling would improve. Although the size of the environment was still limited to the memory capacities of a single PE, the speed up factor possible overrode that consideration. Furthermore the amount of communication involved would still be relatively minimal, consisting mostly of small information messages and large form factor messages which are sent to the master when the form factors for a particular patch are computed. This scheme was the first one that was implemented.

In order to get around the limited environment size the previous scheme can be extended to allow for arbitrarily large environment sizes as long as they fit within the memory of the entire pool of PEs. The basic idea is to distribute the environment among a number of PEs. Each set of these PEs would be termed a cluster. The task farm system would be implemented on top of these clusters. In the previous scheme each cluster contains only one PE, in this scheme each cluster could contain more than one PE. The environment would be distributed across the PEs, hence larger environments could be used. The complexity of the implementation is quite great because the master-slave communication is complicated by the intra-cluster communication. The environment size limit is effectively extended to the size of the machine memory, but the communication requirements in the cluster itself would jump dramatically due to the need to reduce the hemicubes. Due to its complexity this scheme was not implemented.

The final MPP scheme that was considered is also an extension of the basic task farm approach. The extension is to create multiple masters also known as sinks. The idea is to distribute the light sources among the sinks, and have each sink perform radiosity using the global set of slaves which generate the form factors. At the end all radiosities would be added giving a complete picture. Although more of the work is done in parallel, the increase in communication will probably out weigh the benefit of multiple sinks. This is because the cost of shooting a patch is very cheap compared to transmitting the form factors or calculating them. This scheme would not be able to contain large environments unless clustering was used.

Two SMP schemes were considered as part of this project. The first scheme involved parallel generation of each hemicube. This would involve each thread projecting a fraction of the total number of patches onto the hemicube. This would be followed by each thread summing a part of the hemicube. This would be relatively easy to implement, although the required synchronisation involved would have probably slowed down the system to a noticeable extent. Furthermore the communication involved, via semaphores, would be relatively cheap except for the number of times per iteration that the semaphores must be acquired. The environment size would depend on the total global memory of the host and the complexity of the implementation would be relatively low. It was decided that the cost of synchronisation would reduce the efficiency of the algorithm to a relatively large degree and hence this scheme was not implemented.

The second SMP scheme used a task farm approach similar to the MPP models. The only difference is that all memory is global, and some data structures which are static in previous schemes are now allocated on a per thread basis. Since all memory is global communication simply consists of a monitor which contains two queues, one for requests and one for ready made form factors. The entire system requires less synchronisation than the implemented MPP scheme, and is significantly faster. Thus the system scales as well as the MPP version, and has the same small communication profile. Its ability to handle large environments depends on the total memory of the machine and the implementation is very simple. This SMP model was implemented and will be discussed in detail later in the paper.

4 Load Balancing, Communication Overheads and Memory Usage

During the implementation of the MPP and SMP radiosity systems the three main concerns were load balancing, the amount of communication overhead that was involved and memory usage. Each of these issues could seriously degrade performance of the system. Hence, they played a major role in the implementation of the system.

Load balancing attempts to maximise the use of all PEs during the run time of the system. If one PE has significantly more work than other PEs and other PEs depend on it, then the single PE will create a bottle neck leaving the other PEs idle a large percentage of the time. By dividing the work evenly among the processors, balancing the load, the PEs will not remain idle, maximising the amount work they do in the allotted time. In all implementations load balancing consists of simple but effective algorithms which will be discussed in detail later in the paper.

Communication overhead is the cost of communication between the PEs during the execution of the code. For the SMP model this is the cost of acquiring and releasing semaphores or conditions. The communication overhead and load balancing determine for the most part how well the system scales to the number of PEs. The greater the communication cost, the worse the scaling. Specifics regarding each of the implementation's overheads are described in the implementation sections of the paper.

Memory usage is a measure of how much memory is required per PE or thread in order for the system to function. This is especially crucial in the MPP model where no virtual memory exists and each PE has limited memory. Data structures need to be designed such that the amount of memory required is minimal. Usually there is a trade off between memory utilisation and the speed of the system. In this case all implementations were optimised for speed, however some steps were taken to minimise memory usage and will be discussed in following sections.

5 Simple Replicated Environments

The first scheme implemented was the MPP scheme using the simple replicated environment model. This consists of a task farm with a master which acts as source and sink, and slaves which are controlled by the master. The computationally intensive task of computing the form factors for a particular patch is assigned to the slaves. The task of shooting the patch and coordinating the slaves is the domain of the master. The environment is replicated on all PEs at the beginning of the execution after the master parses the input files. Because the environment is replicated, each slave can compute the assigned set of form factors independently of the other slaves. This requires no communication between them. This scheme works quite well for several reasons. The first reason is that the slaves are almost never idle. Although both the master and the slaves take $O(n)$ time to perform one iteration, the coefficient in front of the n for the slave is smaller by a factor of at least 10^2. Thus the master can literally shoot hundreds of patches in the time that it takes to generate one form factor. From these rough calculations it is clear that in most cases the master will be waiting on the slaves to complete a form factor and deliver. The result is that the slaves can be serviced extremely quickly, i.e. assigned another patch for which form factors need calculating. Even if there are several hundred slaves computing form factors the master should be able to keep all slaves busy, maximising PE utilisation.

The system is designed so that form factors which might be needed in the near future are also calculated. The system uses a three level request method by which the master indicates what each slave should do. The first is a request by the master for a certain patch from the slave. If the form factors are already in the slave's form factor cache, they are simply returned to the master. If the form factors are not in the cache, then the request is placed at the front of the slave's to-do queue, and the slave will calculate those form factors during its next iteration. The second level indicator is a suggestion to the slave. During the shooting of a patch the master collects data on patches which might be shot in the near future. These patches are placed into a suggestion queue and are passed on to the slaves when they communicate with the master after each of their iterations. The slaves place these suggestions in the back of their to-do queue and calculate the form factors when the entry is reached in the queue. This is the most common form of communication especially as the execution proceeds, the suggestions become more and more accurate, so a lot of the time the form factors have already been computed by the time they are needed. It is not uncommon for the system to increase in speed in the last quarter of its execution by 10 percent. The third level is to simply do a random patch. The slave selects a patch at random and calculates the form factors for it. This occurs rarely but is useful because eventually those form factors will be used.

The main radiosity system is kept completely separate from the code which performs communication between the master and slaves, and coordinates the computation. The parallel part of the system is kept within a separate module and is accessed through a set of system independent calls by the radiosity system. This creates the ability to attempt different parallelisations without modifying the radiosity system itself.

5.1 Load Balancing

In order to achieve effective load balancing a mechanism had to be created which mapped each patch to a particular slave. The ideal situation is that slaves only compute form factors which are needed in the near future. This would allow all slaves to remain busy and re-use the cache memory efficiently, with a low probability that the form factor currently just destroyed might be needed in a short time. A simple and effective algorithm is to map the patches using a cyclic distribution. The principle is that most patches are stored consecutively in memory if they are adjoint to one another in the environment. Thus of a light source, will occur consecutively in memory and will be distributed among the slaves evenly. Since light sources will be the brightest patches, they will be the first ones that are shot. If we define a light source to be the brightest areas in the environment at time t, then these areas will contain patches which are usually consecutive in memory and will be efficiently distributed among the slaves, and hence will be quickly computed for the $t + m$ iteration where m is the number of time necessary to compute the form factors. Therefore the cyclic mapping provides an effective load balancing strategy.

5.2 Communication

The amount of communication between the master and the slave is kept to a minimum and consists of two types. The first type of communication is a message. The message consists of a message type identifier and a data field. Most of the messages deal with events which occur between the master and the slave. These messages include notification messages that a set of form factors have been calculated or that a set of form factors have been destroyed because the cache had to be reused. The slave can also send messages which tell the master that it is ready to send the form factors. On rare occasions when a slave runs out of things to do, it will send a request for instructions. The master sends complimentary messages like requests, suggestions, and termination messages. All messages are usually the size of two integers and are relatively quick to send. The second form of communication occurs when a request has been made for form factors and the slave sends the form factors. The process involves the following protocol. The master sends a request for a particular set of form factors. If those form factors have not been computed, the slave responds with an appropriate message and places the request at the head its to-do queue. Otherwise the slave sends a ready to transmit message. The master having received this message places itself into a receive state. while the slave having sent the message, sends the form factors as a contiguous array of values. The size of this communication is nf

where n is the number of form factors and f is the size of the value, usually a floating point, on the architecture. This communication, usually on the order of megabytes, is significantly more expensive than the message sending. However, compared to the time taken to calculate the form factors this communication is relatively cheap. Therefore the overall amount of communication does not present a large overhead in the operation of this scheme.

The scheme uses non-blocking synchronous communication to allow the maximum utilisation of the PEs. All slaves block after the send or receive operation, treating it from their end like a blocking synchronous operation. However the master needs to be able to set a receive in motion, check if anything has been received, and if not, do some useful work like shooting patches or making suggestions. If the master used blocking synchronous communication, then it would be forced to not service all waiting slaves, but perform other duties and then service the next slave. Although a probing mechanism could be used to check if any slaves are waiting to be serviced, the non-blocking synchronous communication makes the entire operation more efficient by initially providing the destination buffer. This improves the efficiency of the entire communication system.

In order to keep track of which form factors have been requested, which have completed, and which are being worked on, bitmaps are kept and updated based on the patch number of the form factors. This ensures that patches are not inserted multiple times into the queue and that they are not suggested multiple times which wastes communication resources. The cache system used by the slaves uses a hash table to locate the patches in the cache, this makes the cost of lookup $O(1)$, and benefits the speed of the system greatly in the final phase of the solution.

5.3 Memory Utilisation

Through careful analysis and coding we were able to reduce the amount of memory needed to store the environment on the slave PEs. In order to improve efficiency we need to make the form factor caches as large as possible. Thus we need to minimise the amount of memory needed to store the environment. A patch in the environment contains several pieces of information, including its vertices, area, centre, normal, emission and specularity. In order to calculate the form factors it is not necessary to know the emission, specularity, or maintain the radiosities and unsent energy values which are used by the solver. Thus we split the patch into two records. The first contains geometric information of the patch necessary for form factor calculation, and the second contains radiosity information necessary for the solver. By splitting the patch into two records, we reduce our initial broadcast cost as well as the amount of memory needed by approximately 30%. This translates into a significantly bigger cache.

This scheme is not designed for handling arbitrarily large environments, therefore, if the environment is too large the system will unceremoniously run out of memory and terminate.

The performance of this scheme is discussed later in this paper, the implementation itself was relatively straight forward with few surprises or difficulties. The most interesting possibility of this system is to replace the cyclic mapping with a simple queue. This would require more memory but the overall code might be simplified to some degree. It would also ensure effective load balancing although the suggestion mechanism would have to be deepened and might be more expensive and complicated than before.

6 Symmetric Multi-Processing Task Farm

The SMP parallelisation scheme targets architectures in which all PEs share the same global memory. Since all memory is shared the problem of distributing the environment does not exist and all attention may be focused of achieving the maximum possible speed up. The observations which were made for the first MPP implementation may also be made here. Based on these observations a similar model was designed for the SMP architectures based on the goal of maximum speed up.

The main observation in this case is that patch shooting is significantly faster than the generation of form factors. Based on this observation we decided to use a task farm for the SMP model as well. The SMP model uses threads, instead of processes, which share all memory except for the stack space of each thread. In this model the main thread would become the master thread and would be responsible for shooting the patches and coordinating the slave threads. The master thread spawns an appropriate number of slave threads before entering the solver and would terminate all slave threads once the solver completes. All communication would be done through the same interface as the MPP version which would keep the main radiosity code simple and straightforward, abstracting the thread specific structures to ensure easy portability between different systems.

6.1 Load Balancing

To maintain good load balancing, a different scheme was implemented from the cyclic mapping used in the MPP case. The scheme uses a request queue and a ready queue. The request queue contain patches for which form factors have been requested, and the ready queue contains patches for which form factors have already been calculated and stored. Whenever a slave thread finishes calculating a set of form factors, it stores the form factors in memory, and adds the patch to the ready queue. Finally it adds the form factors to the hash table which is keyed on the patch. It then gets the next patch from the request queue and proceeds to calculate the form factors for it. If there are no request patches available, something that should never happen, the thread waits on the queue's condition variable. The master thread operates in a similar manner. Whenever it finishes shooting a patch, it first checks if the request queue has gotten too small. If so, the master slave adds several suggestions to the request queue to make sure that all slave threads are kept busy. The master thread then checks if the form factors for the next patch to be shot have been completed. If this is so, then the form factors are found in the hash table and used to shoot the next patch. Otherwise the patch is added to the request queue and the master thread checks if there are any ready patches available in the ready queue. If there is a ready patch, then it is removed, the form factors found in the hash table and the patch shot. If there are no patches in the ready queue, the master thread will wait on the queue's condition variable until there are ready patches available.

This scheme ensures that all the slave threads are always working. The master thread will only do as much work as necessary before releasing the CPU to other processes, thus by using n slave threads on an n PE system, maximum utilisation will occur. Since communication is extremely cheap, the system scales up extremely well.

6.2 Communication

In the SMP model, all memory is shared. Hence in order to prevent destructive interference between threads, semaphores and conditions are used in code which operates on shared data structures. Since data structures are shared the communication overhead of sending form factors between threads or passing requests, only depends on the time it takes to acquire the appropriate semaphore. Thus all communication takes constant time and depends on how efficiently the threading libraries implement the semaphores and conditions.

To simplify the design of the entire SMP scheme, the parallelisation part of the code acts like a monitor. All the operations inside the monitor are $O(1)$ in nature and most are no more than 20 to 50 lines of code. The queues contain conditions which allow a thread to release the monitor semaphore and wait for the queue to be filled if the queue is empty when the thread attempts to remove the next element in the queue. Overall the entire system is extremely simple and communication costs are extremely small.

6.3 Memory Utilisation

The global memory model of the SMP architecture allows us to use the entire memory of the machine, including possibly virtual memory, to contain one copy of the environment. Thus we do not need to take any special actions in order to handle large environments, because the system will usually do it for us. The only duplicate memory is the Z-buffer of the hemicube which each thread must have in order to be able to calculate its own form factors. Depending on the size of the hemicube and the number of threads, the memory utilisation for hemicubes alone can grow to tens of megabytes. Apart from this one fact the memory utilisation of the system is extremely efficient and requires a lot less complexity to handle arbitrarily large environments than its MPP counterpart.

7 Analysis of Results

Two distinct effects result when the radiosity algorithm is parallelised. The first is the linear speed up of the algorithm, and the second is the degradation of the resulting image. The linear speed up is the result of throwing more CPU power at the problem. The more PEs we add, the faster the algorithm will run. The second effect is due to the fact that the order in which the patches are shot changes when the algorithm is parallelised. This order changes from a sorted order to a mostly sorted order. Therefore less light is shot during a parallel run of the algorithm than a serial run, for the same number of iterations. The result is that the image generated by the parallel algorithm is slightly degraded when compared to the one generated by the serial algorithm. The speed up of the parallelisation schemes will be discussed first. This will be followed by the analysis of image degradation.

7.1 Speed Up Due To Parallelisation

There are three things that prevent ideal scaling in any of the parallelisation schemes. The first is the I/O which is done at the beginning and the end of the execution. The amount of I/O scales as $O(n)$ where n represents the size of the environment. As the environment size increases, the I/O takes proportionately less time since the main algorithm takes $O(n^2)$ time. As the number of processors increases the I/O will take proportionately more time, because the main algorithm takes proportionately less time to run. Since our concern is the main algorithm, rather than the preprocessing and post-processing steps, we are not too concerned about the effect of I/O on the performance of the code.

The second problem which prevents ideal scaling is the speed of the master process. Depending on how efficiently it and the slave processes are implemented, the master process is probably capable of servicing between 10^2 and 10^4 slaves. Past this point some slaves will be blocked significantly more time than before, creating a reduction in the speed up factor. Currently this is not a pressing concern, but as MPP machines become bigger, the role of the master might have to be delegated to multiple PEs to ensure good speed up.

The last issue that adversely affects speed up is that the number of processes on which parallelisation has been done is actually $n - 1$. Out of a total of n processes, one is the master which spins, and the other $n - 1$ are the slaves. This means that for low numbers of processes, between 2 and 4, the speed up factor is significantly lower.

Below are the run times in seconds for test runs of the system on the CRAY-T3D.

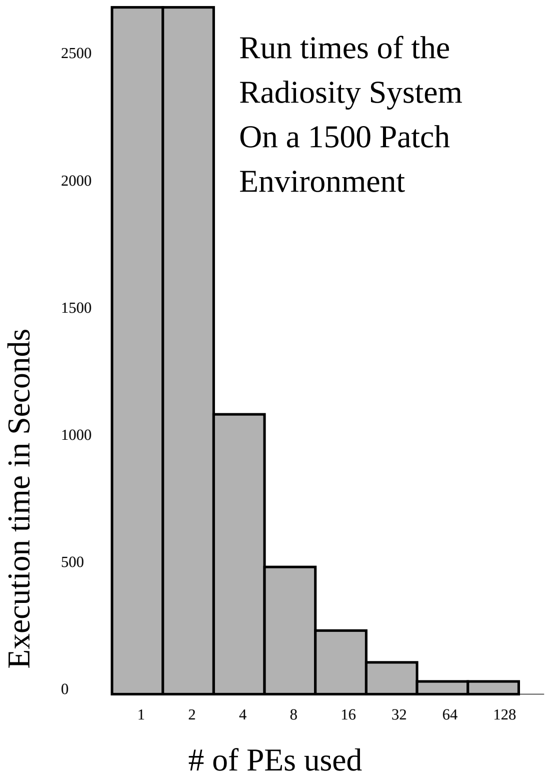

Figure 1: Run-times of the radiosity system on a 1500 patch environment

The run times given are for the calculation stage of the code. The input and output take 2 seconds for the 1500 patch environment and 40 seconds for the 40000 patch case. The environment replication takes less than a second for the 1500 patch case and under 10 seconds for the 40000 patch case.

PEs	1	2	4	8	16	32	64	128
1500 Patches	2700	2700	1090	500	270	140	80	60
40000 Patches	17700	17700	N/A	N/A	1235	866	717	719

Table 1: MPP speed runs on the CRAY-T3D.

The 1500 patch environment is so small that past 8 PEs the serial part of the algorithm, namely the solver affects the results due to the small number of patches that each PE has to calculate. The algorithm becomes limited by the speed of communication and the shooting of patches because in the later stages of the algorithm all the patches have been shot.

Unexpectedly for large environment the system also slows down significantly at around 32 PEs. This cannot be explained at the current time especially since the 16 PE case show an extremely good speed up when compared with the serial run of the algorithm. The only possible explanation is that there is either a bug some where in the algorithm or a significant design flaw which has not been found.

The SMP results are given below. The only machines available for testing were the 2 PE Dec Alphas. Therefore the our speed up measurements were limited. The best time was achieved, as expected, with three threads, by having the two slave threads using the two PEs and giving way to the master thread for a short time to shoot the patches. The results are given below for two different environments.

Threads	1	2	3	4
1500 Patches	640	650	400	430
40000 Patches	1980	1980	1150	1200

Table 2: SMP speed runs on a 2 PE Dec Alpha

7.2 Image Degradation Due To Parallelisation

As the number of PEs or threads increases, the image being generated degrades slightly due to the fact that the ordered approach to shooting patches is replaced by a partially ordered approach. To this end, as the number of PE's increases, the number of iterations should also be increased. To determine the amount of image degradation, we compared all parallel generated images to a serially generated image. During each comparison we note two things. The maximum difference between the two images, which is simply the greatest different between the the pixel of one image and its counterpart in the other, and the average difference which involves summing up all the differences between pixels and their counterparts, and dividing by the number of pixels. These results are listed in the table below. We only used the 1500 patch image to determine the general progression of the degradation. The degradation should be relatively independent of the environment size since the number of light sources in any environment is relatively small. To calculate the difference we take the magnitude of the pixel's brightness and divide by the maximum magnitude of brightness possible. This is given by the following equation:

$$\Delta_p = \sqrt{\frac{R_d^2 + G_d^2 + B_d^2}{3 * 255^2}}$$

Where R_d, G_d and B_d are the maximum or average differences of the pixel colours in the range of 0 to 255.

PEs	1	2	4	8	16	32	64	128	256
Max Δ	0	0	7.1	9.9	18.4	13.3	19.4	20.5	3.8
Ave Δ	0	0	0.4	0.6	1.8	2.4	3.3	0.9	0.4

Table 3: MPP degradation runs, differences in percent.

Due to time limits the degradation of image quality was not fully investigated and is left as future work. For this paper one environment consisting of 1500 patches was used with varying number of PEs. The observations, listed in the table above, indicate that the maximum difference between a serially generated and a parallel generated image levels out at 16 PEs at around 20%. What is even more promising is that the average difference in most case is no more than 4% in this case. This indicates that the number of iterations will only have to increase slightly as the number of PEs increases to ensure the same amount of accuracy as generated by the serial algorithm.

By replacing the cyclic mapping of patch to PE with a queue, this difference might go down even more because the request order, which would be sorted, would imply a nearly identical form factor generation and utilisation order. This should drop the differences to almost zero and bares further investigation.

8 Conclusion

The main goal of this investigation was to investigate and implement efficient parallelisation strategies for the radiosity technique. The motivation for this investigation is based on the high degree of realism that can be generated using radiosity, which is tempered by the large computational costs of the algorithm. The main approach consisted of identifying parts of the algorithm which were expensive as well as easily parallelisable, and implementing the parallelisation with the credo of keeping the design simple. The target architecture under investigation were MPP and SMP. In both cases similar designs were used.

Our main assumption was that the process of shooting patches requires significantly less time than the process of generating the form factors. Based on this assumption we used a task farm model in all three designs that were proposed. The three proposed designs were: The task farm with a replicated environment, the task farm with a distributed environment and the SMP version of the task farm. In all three cases one PE, cluster or thread was responsible for shooting the patches and was considered the master. The rest of the PEs, clusters or threads were considered slaves and had the principal task of generating form factors. Each of the designs had its set of attractors and detractors.

The task farm with the replicated environment, was designed for the MPP architecture and had excellent scalability, little communication and simple design. For small environments it was the ideal model. The main drawback was the fact that it could not handle large environments which would become more prevalent as the scenes increased in complexity.

To try a different approach we implemented the task farm using the SMP model. This reduced our communication to nearly zero, removed the problem of distributing the environment and simplified complexity to the bare minimum. The only drawback is that most SMP architecture have no more than 16 PEs which means that the scaling factor is only good up to 17. Thus the linear speedup is limited. However the number of SMP machines is significantly more than the number of MPP machines. By using several SMP machines and letting each one shoot a sub set of the light sources, equivalent power of a MPP architecture can easily be reached.

The performance of the implemented parallelisation schemes was generally acceptable. The image degradation expected to occur when significant number of PEs were used levelled out at 16 PEs. Thus by using more PEs and increasing the number of iterations slightly, significant accuracy in a reasonable time can be achieved. Along the way several interesting topics of investigation were discovered.

9 Future Work

There are many things that still need to be investigated surrounding this topic of interest. While some of the items are purely mechanical others present algorithm design challenges.

9.1 Improving the Main Algorithm

Currently the radiosity algorithm does not take into account the distance between patches. As light propagates it disperses. The result is that weak light does not reach patches which are of significant distance away, or if it does, the effects are inconsequential. Therefore a patch culling system may be put in place which does not bother to calculate form factors between patches if very little light is every going to be exchanged, i.e. between two patches which are at the baseboards of the opposite sides of a room. It would be interesting to investigate under what parameters culling should occur. This may reduces the algorithmic run time of generating form factors to some degree.

9.2 Implementing the Distributed Environment Task Farm

The MPP scheme implemented does not address the problem of arbitrary large environments. However if we wish to handle large environments which contain millions of patches, it would be prudent to implement and investigate the operations of such an scheme. Due to the complexity of such a system it could take significantly more time to implement.

9.3 Environment Partitioning

In order to accomplish the previously mentioned task an algorithm for partitioning the environment over multiple PE's must be developed. The one such algorithm was already outlined in the paper, but the implementation across multiple PEs is still to be worked out. This in itself would present an interesting research topic and would be applicable to more than radiosity.

9.4 Non Cyclical Mapping of Patches to PEs

In the first outlined implementation a cyclical mapping from patch to PE was used to create a natural load balancing. Although this will work well in the average case, there are worse case scenarios which are not promising. Other mapping techniques, like queueing, should be investigated.

9.5 Sub-structuring

In order to make the serial radiosity algorithm more efficient and generate better looking images, a dynamic sub-structuring technique can be applied to divide certain patches into smaller elements[4]. Implementing this technique in a parallel environment could present its own problem with communication and synchronisation. However implementing such a technique could reduce run time and create better images. Thus further research into this topic is strongly recommended.

9.6 Image degradation

The analysis of image degradation in this paper is very rudimentary and simple. More observations should be made with different sizes and types of environments. Furthermore different algorithms might improve or worsen the image degradation experienced wit the afore mentioned parallelisation schemes. The main problem is to ensure patch order, while at the same time maximising the speed up of the system.

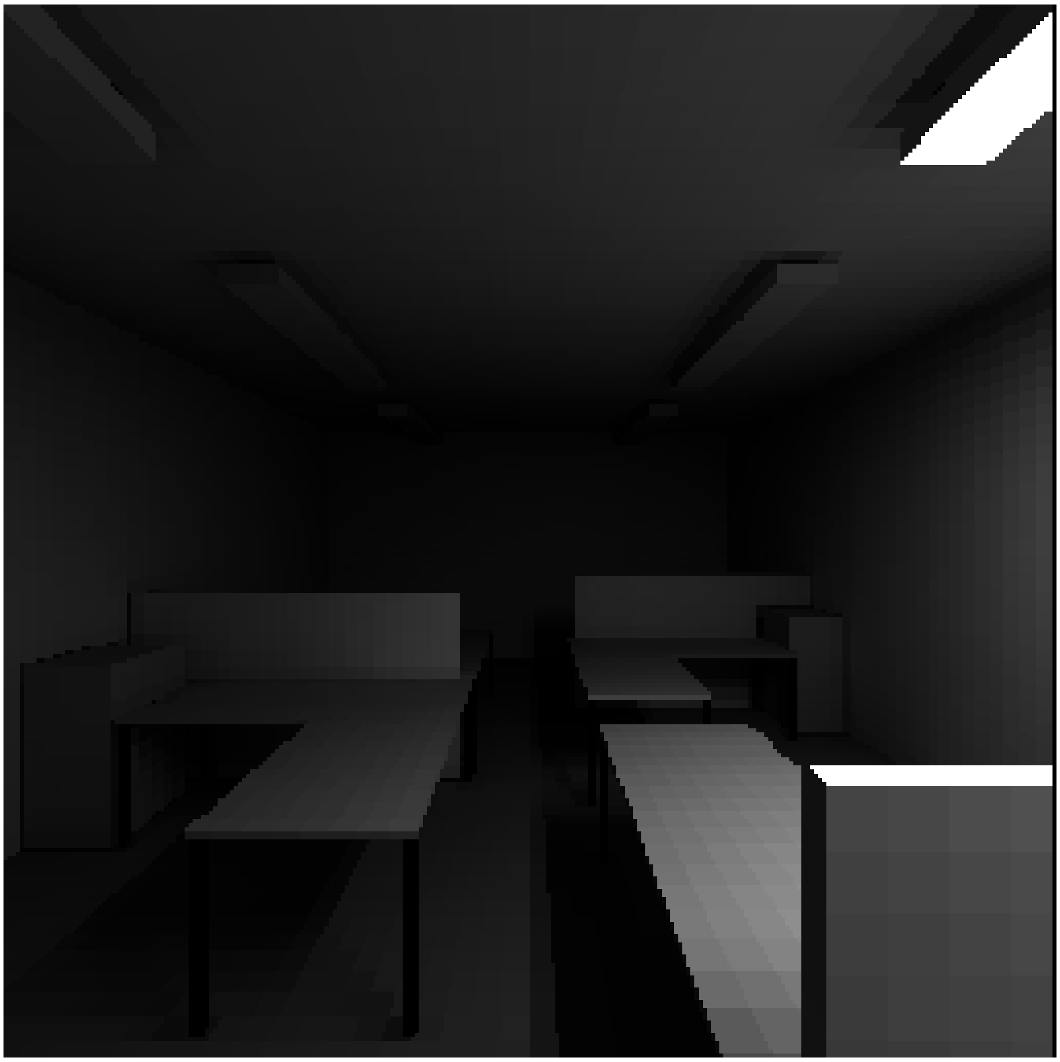

Figure 2: Image generated using the MPP parallel radiosity system

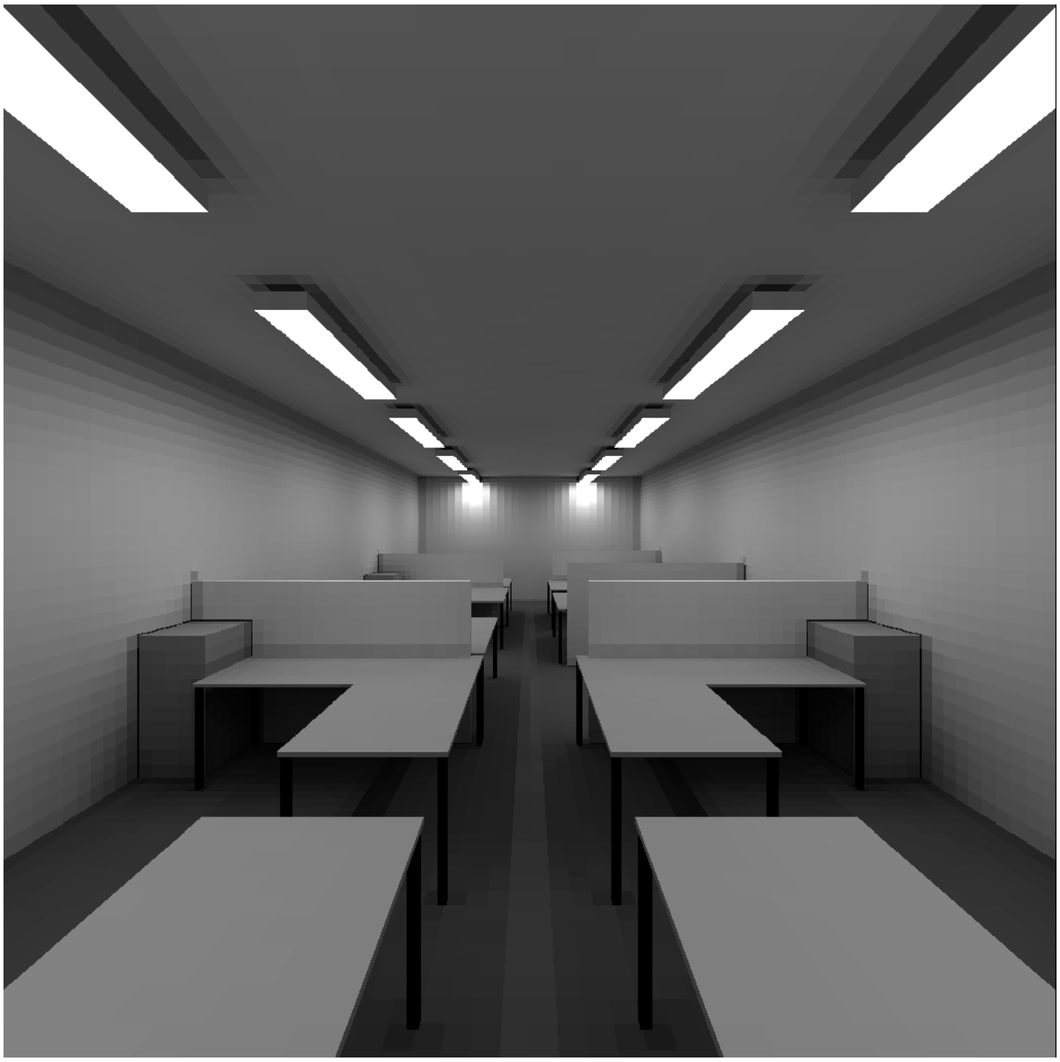

Figure 3: Image generated using the SMP parallel radiosity system

References

[1] D. Baum and J. Winget. Real time radiosity through parallel processing and hardware acceleration. *Computer Graphics*, 24(4):67–75, 1990.

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

[3] M. Cohen and D. Greenberg. A radiosity solution for complex environments. *Computer Graphics*, 19(3):31–39, 1985.

[4] M. Cohen, D. Greenberg, and D. Immel. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics and Applications*, 6(2):26–35, 1986.

[5] N. Gatenby. A proposed alternative to the hemicube algorithm. Master's thesis, University of Manchester, 1990.

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

[7] D. Immel, M. Cohen, and D. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics*, 20(4):133–142, 1986.

[8] R. Siegel and J. R. Howel. *Thermal Radiation Heat Transfer*. Hemisphere Publishing, 1984.

[9] Alan Watt. *3D Computer Graphics*. Addison-Wesley Publishing Company, 2 edition, 1993.