PARTITIONING AND SCHEDULING LARGE RADIOSITY COMPUTATIONS IN PARALLEL*

XAVIER CAVIN , LAURENT ALONSO , AND JEAN–CLAUDE PAUL

Abstract. We show, in this paper, how it is feasible to efficiently perform large radiosity computations on a conventional (distributed) shared memory multiprocessor machine. Hierarchical radiosity algorithms, although computationally expensive, are an efficient view–independent way to compute the global illumination which gives the visual ambiance to a scene. Their effective parallelization is made challenging, however, by their non–uniform, dynamically changing characteristics, and their need for long–range communication. To address this need, we have developed appropriate partitioning and scheduling techniques, that deliver an optimal load balancing, while still exhibiting excellent data locality. We provide the detailed implementation of these techniques and present results of experiments showing very good acceleration and scalability performances. The accurate radiosity solutions required to render high quality images of an extremely large model are computed in a reasonable time. The rendering capabilities of modern graphics hardware are then used to visualize this virtual pre–lit environment in real–time: a two minutes QuickTime movie example can be downloaded from our site: `ftp://ftp.loria.fr/pub/loria/isa/Cavin/sodaHallWalk.qt.gz`.

Key words. parallelism, data locality, load balancing, computer graphics, radiosity

AMS subject classifications. 68U05,68Q22,65Y25,65Y05

1. Introduction. The computation of radiosity, power per unit area $[W/m^2]$, is widely used in computer graphics to simulate the distribution of light in a three dimensional environment, and to render impressive realistic images. Radiosity algorithms [7, 18], based on finite element methods, interleave the computation of global light transport and local lighting representations in a given scene. Using the typical assumption of diffuse reflectors, these approaches model the transport process by determining the property of illumination leaving one surface element and reaching another. A major advantage of the radiosity approach is that, once the scene illumination is computed, the results are independent of the observer position. Thus, realistic simulations can be displayed in real–time using standard graphics hardware accelerators, since the lighting computations do not have to be repeated. Radiosity algorithms suffer, however, from a number of problems which have restricted their spread and use in real–world applications. One of the most important limitations may be their lack of capability to perform high quality computations on large complex models.

Typically, in this application, the volume of data depends obviously on the input data, but also on the output information. The radiosity integral equation to be solved relies on an explicit discretization of the illumination function, reducing the computation to the resolution of a linear system, with coupling coefficients: the number of elements required to approximate the radiosity solution can be much greater than the input data. Moreover, the auxiliary information generated temporarily by the algorithm to compute the underlying coefficients — visibility queries in particular — is not known *a priori*, and can be even greater than the output information. These limiting factors in performing large simulations make the computation of accurate radiosity solutions for complex models impractical on single processor platforms.

The parallelization of a hierarchical radiosity application is challenging, due to its non–uniform and dynamically changing characteristics, and its need for long–range communication. The non–uniform characteristic is intuitive: the distribution of light

*LORIA – INRIA Lorraine, Campus Scientifique, BP 239, F–54506 Vandœuvre–les–Nancy Cedex, France ({`cavin,alonso,paul`}`@loria.fr`).

interactions between surfaces in a given scene is highly irregular. Its dynamic nature relies on the hierarchical formulation: in hierarchical radiosity, a very deeply nested hierarchy of mesh element subdivisions can be imposed on every initial surface. Light transport is allowed to occur throughout these hierarchies. As opposed to classical algorithms using a quadratic number of interactions, hierarchical formulations considerably improve the performance of radiosity computations, but change their characteristics at every step of the resolution. Since two mesh elements can interact at any level of their hierarchies, sub–computation times are highly variable, making load balancing difficult. Moreover, since both emitting and receiving surfaces can be dynamically subdivided, reader/writer locks must be used to enforce concurrency control during updates, and deadlock avoidance/resolution must be considered. As shown clearly in [20], straightforward decomposition techniques that an automatic scheduler might implement do not scale well for N–Body methods. Since hierarchical radiosity typically applies some aspects of these methods, it is unable to simultaneously provide load balancing and data locality, so appropriate partitioning and scheduling techniques need to be designed.

In this paper, we address this problem with the parallel implementation of a hierarchical wavelet radiosity algorithm on a SGI Origin2000. More precisely, we give a detailed description of techniques which yield very good speed–ups and memory cache hits, allowing the computation of hierarchical radiosity solutions on an extremely large–scale test model (Figure 6.1), with both high accuracy and reasonable speed considerations. We present in the next section the computational cost aspects of the radiosity application and existing parallel radiosity implementations to address this problem. In §3, we focus on the available parallelism of the radiosity algorithm, while techniques to perform load balancing and data locality are detailed in §4. Section 5 contains results and discussion of our experiments. Finally, §6 concludes and presents some directions for future works.

2. Radiosity computation considerations.

2.1. Solving the radiosity equation. Radiosity is governed by a Fredholm integral equation of the second kind. It arises from a more general equation known as the rendering equation [15], when one assumes that all reflections occur isotropically. Given some physical assumptions, the radiosity equation can be formulated as follows:

$$f(x) = g(x) + k(x) \int_{\mathcal{M}} \frac{\cos\theta_{x'} \cos\theta_x}{\pi r^2_{x'x}} \, v(x', x) f(x') \, dx' \tag{2.1}$$

where $k(x)$ is the local reflectance function of the surface, θ_x and $\theta_{x'}$ are the angles between the surface normals and the vector connecting the two points x and x', $r_{x'x}$ is the distance between theses two points, and $v(x', x)$ is the visibility function whose value is in $\{0, 1\}$ according to whether the line between the two points x and x' is respectively obstructed or un-obstructed.

Finding an analytical solution to this equation is not possible in general. Numerical approximations must be employed, generally leading to very expensive algorithms. The fundamental reason for the high cost of numerical approximations is that each surface in a given scene can potentially influence all other surfaces *via* reflection.

A resolution technique for integral equations such as (2.1) is the weighted residual method, often referred to as "finite element method". Previous classical radiosity algorithms [11, 6] can be analyzed as finite element methods using piecewise constant basis functions. Galerkin radiosity, first introduced by [14, 22, 24], aims to increase the order of basis functions used in radiosity algorithms. Their goal is to improve the

quality of the computed solutions, as well as the efficiency of the computations. Thus, using higher order basis functions allows the use of much coarser meshes than classical radiosity, while still meeting a requested error bound. Hierarchical formulations of radiosity algorithms have been introduced by [5, 13]. These formulations consider the possible set of interactions in a recursive scheme, allowing the number of interactions to be linear. Wavelet radiosity, introduced by Gortler *et al.* [12] and Schroder *et al.* [17], unified the benefits of higher order Galerkin radiosity and hierarchical radiosity. The algorithm presented in this paper relies on this approach.

2.2. Previous parallel radiosity implementations. Most parallel implementations of radiosity [4] relied on the classical algorithm, which has quadratic time complexity. In [9], Funkhouser has described a "system for computing hierarchical radiosity solutions for very large polygonal models using multiple concurrent processes". The proposed solution is a parallel version of the partitioning and ordering techniques proposed by Teller in [21], which are strongly dependent of the axial and densely occluded geometry of the model. The Soda Hall test–model used in this paper is, up–to–now, the most complex scene, in terms of input surfaces, for which a radiosity solution has ever been completed: 280 836 were used to model 250 furnished rooms of five floors of the building, for a total area of 75 946 664 square inches. The total execution time was 168 hours, using an average of 4.96 slave processors.

As pointed out by the author, the implementation and analysis of a distributed approach to the radiosity problem requires careful consideration of group partitioning, data distribution, and load balancing issues. Coarse–grained parallel execution, based on multiple separate copies of a shared database, allows multiple processors to execute concurrently with little contention or synchronization overhead. However, since updates to the shared database are executed with a coarse granularity, many of the sub–computations may be performed using out–of–database values, potentially reducing the convergence rate. This was a strong limitation to the scalability of this parallel algorithm.

Up to now, as argued in [20] and experimentally proven in [10], it appears that due to its communication nature, large hierarchical radiosity computations are not practical for a network of workstations. Some authors have implemented a parallel radiosity solver for a shared–memory multiprocessor system. Singh *et al.* [19] have experimented a very fine-grained parallelization of hierarchical radiosity on the 48 processors Stanford's DASH machine: using distributed task queues with a task stealing mechanism, they were able to obtain an optimal load balancing, while showing that the cache coherent shared address space architecture was particularly well adapted to handle the natural data locality exhibited by this kind of algorithm. Their experiments have yielded very good performance, but on a very small test model (174 input polygons). We think, however, that when dealing with much larger environments, the amount of communication certainly changes the dimension of the problem. Despite this restriction, [20] greatly influenced our work. We must also mention the parallel implementation proposed in [16] on a SGI Origin2000 machine, which aimed to optimize data locality. Unfortunately, the proposed technique is only valuable for classical radiosity algorithms.

In summary, all previous parallel versions of radiosity algorithms failed to *simultaneously* perform:

1. large radiosity computations [19, 20];
2. with a linear time algorithm [16];
3. with high order basis functions [20, 9];

4. with both an optimal load balancing and an effective data locality [16];
5. with a moderate scalability [9].

This paper addresses all these limitations.

3. Issues to effective parallelism.

3.1. Sequential algorithm. The sequential algorithm we consider is a wavelet based hierarchical radiosity algorithm, with several improvements in calculation proposed in [23]. Basically, the algorithm proceeds as follows (algorithm 1): first, the radiosity function B_x is initialized over every input surface $S_x \in \{S_1, \ldots, S_n\}$, either from the point light sources direct illumination computation phase, or with its self–emitting radiosity, or with the zero function; simultaneously, we compute an estimate $\mathcal{E}_x$ of the corresponding unshot energy, and maintain a sorted list of input surfaces, by decreasing unshot energy order.

Then, successive *shooting iterations* are iteratively performed to update the surfaces' radiosity function. At the beginning of each shooting iteration, the surface S_e with the largest unshot energy is removed from the sorted list. Its residual radiosity is successively propagated to every visible surface S_r of the scene: for each *interaction* between the *emitting surface* S_e and a *receiving surface* S_r, a *hierarchical radiosity transfer* is computed to update the radiosity function B_r. A part of the radiosity received by S_r is absorbed and the other part is reflected, and thus incorporated into the residual radiosity of S_r: this increases the unshot energy $\mathcal{E}_r$ of S_r, whose position in the sorted list has to be updated. A shooting iteration is completed after all energy transfers between S_e and its receiving surfaces S_r are completed. Finally, the unshot energy and residual radiosity of S_e are reseted, and the next shooting iteration begins, unless the desired convergence rate[1] has been reached.

Algorithm 1 Sequential algorithm.

```
1: for all S_x ∈ {S_1, ..., S_n} do
2:    B^x ← E^x
3:    ℰ_x ← Energy(B^x)
4: end for
5: while not converged do
6:    Choose the surface S_e with the largest unshot energy ℰ_e
7:    Push/Pull(S_e)
8:    for all S_r ∈ {S_1, ..., S_n} − S_e do
9:       HierarchicalRadiosityTransfer(S_e, S_r)
10:      ℰ_r ← UnshotEnergy(B^r)
11:   end for
12:   ℰ_e ← 0
13: end while
```

The hierarchical radiosity transfer is based on a multi-level representation of the radiosity function over the surfaces. Each surface is represented as a *quadtree* of *mesh elements*, over which the radiosity function is projected onto wavelet basis functions. The radiosity transfer starts at the higher level of the quadtrees. A user–parameterizable *oracle* function decides if the transfer can be done between the two current mesh elements, based on an evaluation of the committed error. If the error is too high, the transfer must be performed between one of the two mesh elements and the lower level representation of the other one (this lower level representation may have to be created). Energy can thus be transfered between any levels of the

[1] Computed as $1 - (total\ remaining\ unshot\ energy/total\ initial\ unshot\ energy)$.

quadtrees. When the radiosity function has to be updated over the whole surface quadtree, a *push/pull* mechanism is used to transmit the energy between its different levels.

The oracle function involves a huge amount of point–to–point visibility computations, which can be accelerated with a *Binary Space Partition* structure [8], or with graphics hardware [1] when available.

3.2. Available parallelism. The sequential parts of our application involve loading the input scene and initializing the data structures, building the BSP, visualizing and exporting the result.

Parallelism can be exploited in the solving phase of the algorithm and is available at several levels:

1. *across emitting surfaces*: each process P gets a different emitting surface S_e^p (algorithm 1:6), and computes the whole set of radiosity transfers towards every receiving surface $\{S_1, \ldots, S_n\} - S_e^p$;
2. *across surfaces' radiosity transfers*: each process computes a single interaction between an emitting surface S_e and a receiving surface S_r (algorithm 1:9), thus several processes can deal with the same emitting surface at the same time (with different receiving surfaces);
3. *across mesh elements interactions*: several processes may collaborate to compute a single radiosity transfer (algorithm 1:9) between the same emitting and receiving surfaces (respectively S_e and S_r).

3.3. The execution platform. In order to fully exploit a computer, it is critical to understand the underlying architecture. The SGI Origin2000 is a scalable multiprocessor system (up to 1024 processors) with distributed shared memory (DSM), based on the SN0 (Scalable Node 0). This architecture provides both the programming simplicity of a shared memory architecture and the scalability of a distributed memory architecture, by eliminating the limited bandwidth of a common bus. For a 64–processor machine, the number of router hops[2] is less than five with an average of 2.97, giving an average read latency of 796 ns (with a TLB hit). However, in order to get optimal performance, it seems necessary for programs to use the caches effectively. The great majority of data accesses should be satisfied from the caches, thus making the access time to memory (local or remote) less important. This can be achieved by applying the two straightforward principles of data locality:

1. *spatial locality*: a program should use every word of every cache line (128 bytes) it touches, to avoid the time wasted copying the unused parts of the line;
2. *temporal locality*: a program should use a cache line intensively, and then not return to it later, because it may have been replaced by other data.

At least, a process should use the memory that is closest to the processor it runs on. Fortunately, the SN0 architecture provides both hardware and software features for improving performance, including support for dynamic page migration (in order to have data pages reside primarily in local memory) and prefetching (so memory–fetch can be overlapped with execution).

4. Performing load balancing and data locality. Exploiting data locality in the extended memory hierarchy of a parallel machine is a key issue in order to both improve single node performance, and to reduce the extra communication between nodes. In addition, we showed in [3] that high parallel performance and scalability

[2] The number of routers that could handle a request for memory data.

could obviously not be obtained without optimally balancing the workload among processes. In [2], we experimented with the three granularities of parallelism, previously mentioned, and showed that there is no hope to sustain good parallel performance with the the coarsest or the finest granularity. We focus on the third remaining approach, which leads to a good load balancing without losing control over the data locality. Two critical problems concerning concurrent access issues are solved here: the duplication of the emitting surface quadtree, and the restrictive access to the receiving surfaces. Finally, memory parallel management considerations are exposed.

4.1. Parallel algorithm. The complete parallel algorithm is described by algorithm 2, with the same notations as introduced in the sequential algorithm 1.

Algorithm 2 Parallel algorithm.

1: **for all** $S_x \in \{S_1, \dots, S_n\}$ **do**
2: $B^x \leftarrow E^x$
3: $\mathcal{E}_x \leftarrow Energy(B^x)$
4: **end for**
5: /* **In parallel: each process performs the same loop** */
6: /* **Note:** $TaskManager$**,** $TM_emitter$ **and** $TM_receiver$ **are shared objects** */
7: **while** not converged **do**
8: $Tmp \leftarrow S_e$
9: $Lock(TaskManager)$
10: **if** $TM_receiver = S_{n+1} \,||\, TM_emitter = \emptyset$ **then**
11: Choose the surface S_x with the largest unshot energy $\mathcal{E}_x$
12: $Lock(S_x)$
13: $Push/Pull(S_x)$
14: $TM_emitter \leftarrow S_x$
15: $TM_receiver \leftarrow S_1$
16: **end if**
17: $S_e \leftarrow TM_emitter$
18: $S_r \leftarrow TM_receiver$
19: $TM_receiver \leftarrow S_{r+1}$
20: $Unlock(TaskManager)$
21: **if** $S_e \neq Tmp$ **then**
22: $LcS_e \leftarrow LazyCopy(S_e)$ /* Since S_e is locked */
23: **end if**
24: **if** $GetLock(S_r)$ **then** /* Receiving surface S_r was not locked */
25: $HierarchicalRadiosityTransfer(LcS_e, S_r)$
26: $MergePendingLures(S_r)$ /* If any */
27: $\mathcal{E}_r \leftarrow UnshotEnergy(B^r)$
28: $Unlock(S_r)$
29: **else** /* Receiving surface S_r is locked, as emitter or receiver */
30: $LuS_r \leftarrow MakeLure(S_r)$
31: $HierarchicalRadiosityTransfer(LcS_e, LuS_r)$
32: **if** $GetLock(S_r)$ **then** /* Receiving surface S_r has been unlocked */
33: $MergeLure(LuS_r, S_r)$
34: $\mathcal{E}_r \leftarrow UnshotEnergy(B^r)$
35: $Unlock(S_r)$
36: **else** /* Receiving surface S_r is still locked */
37: $AddLureToPendingLures(LuS_r, S_r)$
38: **end if**
39: **end if**
40: **if** $S_r = S_n$ **then** /* Last radiosity transfer of the emitting surface S_e */
41: $\mathcal{E}_e \leftarrow 0$
42: $MergePendingLures(S_e)$ /* If any */
43: $\mathcal{E}_e \leftarrow UnshotEnergy(B^e)$
44: $Unlock(S_e)$
45: **end if**
46: **end while**

Basically, its principle is to *continuously* process the radiosity transfers of the successive shooting iterations "in nearly the same order" as in the sequential algorithm. Actually, no synchronization is done at the shooting iteration level (algorithm 2:11): some processes can begin the following shooting iteration without waiting for the other processes to complete the current shooting iteration. They use the surface *currently* in the first place of the sorted list. So, this may not be the same emitting surface as if they had waited for the current shooting iteration to be completed, but this avoids having them remain idle on a synchronization barrier.

Obviously, we had to introduce some mechanisms to ensure safe parallel computations of radiosity transfers:

1. Due to the parallelization scheme, an emitting surface is used by multiple processes. These processes may have to create previously non–existent subdivisions of mesh elements for accuracy considerations. In order to avoid a locking mechanism overhead, and because the radiosity of the emitting surface is not affected by these subdivisions, the processes can use separate copies of the emitting surface.

2. Because there is no synchronization at the shooting iteration level, a surface can simultaneously receive energy from different emitting surfaces. A receiving surface must be locked during a given energy transfer to ensure the coherency of its radiosity function.

3. Furthermore, a given surface can be used as a receiver and an emitter at the same time. On one hand, a receiving surface can unfortunately not be used as an emitting surface, since the radiosity values stored on its mesh elements are being updated. On the other hand, an emitting surface keeps its radiosity values coherent: a solution can be found to use it as a receiving surface, as long as its receiver role does not affect its emitter role.

The first point, copying the emitting surface, introduces a parallelism overhead, which may be problematic, especially at the end of the computation: the duplication of a large, deeply decomposed surface, may take longer than computing the corresponding shooting iteration. The second and third points may lead to severe lock problems. On one hand, an emitting surface cannot receive a radiosity transfer until all its receiving surfaces have been processed, which may take time. On the other hand, simultaneous radiosity transfers towards the same emitting surface are serialized, thus idling some processes. Moreover, a deadlock situation is likely to occur, when two surfaces are involved in symmetrical energy transfers. These two issues are respectively solved with the introduction of our *lazy copy* and *lure* original techniques.

4.2. Lazy copy. Copying the emitting surface appears to be a critical point in our parallel algorithm, due to the following:

1. emitting surfaces are more and more subdivided, thus increasing copy time, while corresponding shooting iteration computation times decrease;

2. through the oracle function, only the higher levels of the duplicated mesh elements hierarchy remain useful, as the computation proceeds.

This naturally leads to the idea of using a light copy of the emitting surface. For building this so called "lazy copy", we no longer duplicate the whole hierarchy, but only its higher level (the root), keeping a reference to the original surface. Then, when the process needs a lower level of the hierarchy, it copies the information from the original surface, if it exists, or creates it on its lazy copy. Low levels of the hierarchy are only copied when necessary. Further subdivisions done on the lazy copy are not copied back to the original surface, since they do not contain supplementary radiosity information.

4.3. Lure. We also had to find a mechanism to avoid that a process remains idle when it has to wait for a receiving surface to be unlocked, and at the same time to resolve the potential deadlock problem. The basic idea is to let the radiosity transfer work on a disjoint simplified copy (*i.e.* without subdivisions) of the locked receiving surface, namely a *lure*. Once the radiosity transfer is completed, the computed lure and the original receiving surface hierarchies must be merged. When a process has completed its radiosity transfer with a lure, the original surface the lure was built from may either be locked or not. If it is no longer locked, the process locks it and can easily perform the merging operation. In the other case, the process should not wait for the surface to be unlocked, or there would be no benefit. It thus appends its lure into a list of "pending" lures associated with the original surface.

We now have to determine how and when the pending lures will be merged with their original surface. This task could be assigned to the next process computing a shooting iteration with this surface as an emitting surface. Another, more efficient solution, is to have it done by the process in charge of unlocking this surface, allowing the merging work to be divided among processes. Moreover, during the merging operation, which may be time consuming, processes requiring this surface as a receiver for a radiosity transfer will still be able to use another lure. If it had been assigned to the process getting the first radiosity transfer for a shooting iteration with this surface as an emitting surface, this would have delayed the following processes assigned to its next radiosity transfers.

To be complete, we have to note that a potential lock idle time problem remains, when a new shooting iteration starts, involving an emitting surface which is currently locked as a receiving surface. This problem may be avoided by introducing a *heuristic*, forcing a process to use a lure when it has to compute a radiosity transfer towards a receiving surface which "is likely to become an emitting surface in the near future".

4.4. Memory management. Efficiently managing the shared memory is a crucial point for such a parallel application which constantly allocates and deallocates memory data. Our early experimentations using the standard memory allocation library have proven that this method reveals itself as a serious bottleneck. Indeed, *each time* a process needs to allocate or free a piece of memory, it must use a lock in order to avoid trashing the allocation table. This severely restricts the expected speed–up.

In order to avoid this bottleneck, and to avoid rewriting all memory allocations within our application, we have chosen to create a memory allocation space for each process (*i.e.* an arena) and to overload the `malloc`, `realloc` and `free` functions, so that each process only allocates data in its own arena. In order to allow allocated data to be freed, we add a 64 bits integer flag in front of each allocated data, in order to code in which arena it has been allocated. A process having to free a given data has to look at this flag to find the arena where it actually performs the free operation. In fact, as a great number of deallocations are done by the process which has previously allocated the data, there is very little contention in practice for accessing the arena.

Our new memory parallel allocation library generally gives good results in term of execution time, since concurrent processes no longer spend most of their time trying to access the main memory to allocate or free data. Nevertheless, in terms of memory space, the results are not so good. This seems to be due to the SGI "fast main memory allocator" (`-lmalloc`), on which the arena main memory allocator (`amalloc`) is based. Indeed, this library manages the dynamic allocation less efficiently than the standard one, and it is not so rare, when doing a call to `amallinfo`, to notice that less than 20% of the space allocated to the ordinary blocks is effectively used in a given arena.

Such bad results are not observed with the sequential application, which uses the standard `malloc`. We are currently trying to modify this behavior, by fixing different combinations of the allocator parameters: as an example, increasing M_NLBLKS seems to help in some ways.

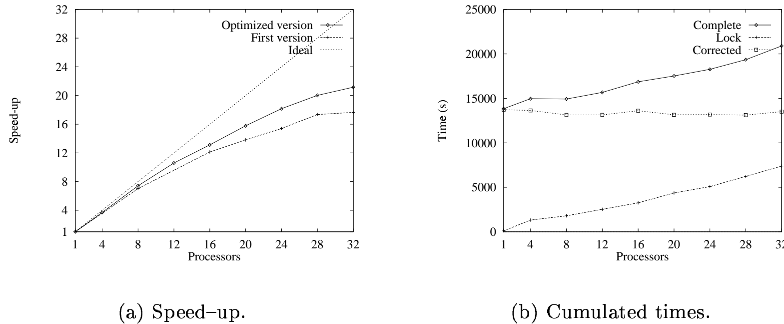
5. Results and discussion. In order to test the effectiveness of our parallel algorithm, we performed several experiments with the well–known Soda Hall database, mentioned in §2. This dataset consists of eight floors, six of which containing furniture, for a total of 558 300 surfaces. For this experiment, we used the SGI Origin2000 described in §3. The machine is equipped with 64 R10000 processors, for a total of 24 Gbytes of main memory. The operating system is the IRIX 6.5.5f release. The parallel algorithm has been implemented within a research application of more than 400 C++ classes, based on the SGI OpenInventor library. We used `sproc` processes, and compiled the application using the MIPSpro 7.2.1 C++ compiler, with the maximum available performance optimization flags.

Evaluating a parallel application is difficult, since many parameters are involved in the overall behavior of its execution. The key question, regarding our problem, is to evaluate the data locality and load balancing properties. Fortunately, the R10000 processors of the SGI Origin2000 provide hardware counters for measuring, among others, memory cache hits, through the `perfex` tool. Characterizing load balancing is more complicated than only measuring complete execution time. Doing this, even in combination with the counters analysis, would hide uncontrolled idle time — such as system activity and hidden system locks — that obviously must be taken into account, but does not characterize the parallel algorithm itself. A more accurate solution is to precisely measure the individual idle time lost waiting on each synchronization point of the parallel algorithm. However, this timing must not introduce a supplementary cost, *i.e.* we shall avoid using system calls: we have chosen to instrument our code by mapping the R10000 cycles counter, allowing an instant, cost–free, measure.

5.1. Performance evaluation. For performance evaluation purposes, we used one *unfurnished* floor of the Soda Hall model. This test model is composed of 8 548 initial surfaces, of which 52 are self–emitting surfaces. Although not so complex, this model was useful for this part of experimentations: using a larger model would have been not so convenient for speed–up measures. Moreover, when the needed memory no longer fits into the main memory of a single processor, some side–effects occur that may enhance the obtained parallel performance. The sequential execution took 10 813 seconds to complete, creating 128 039 mesh elements, 98 176 of which are final. 1 086 shooting iterations were needed to reach a 90 % convergence rate.

The obtained data locality is very high, since 95% of data accesses are satisfied in both $L1$ and $L2$ caches, independently of the number of processors used. These results are very similar to those obtained in [16], where an explicit data distribution was used to optimize data locality. This confirms both our parallelization design choices, and the capabilities of the ccNUMA architecture to efficiently handle the inherent data locality of our application. The lower curve of Figure 5.1(a) shows a plot of speed–up for increasing numbers of processors, from 1 to 32, with a first implementation. Although not linear, the obtained performance is not so bad, in regards to the high convergence considered, since we obtained more than 50% of linear speed–up up to 32 processors. By examining our synchronization points instrumentation figures, we can notice that our load balancing strategies are excellent: no idle time is lost, neither on the unique final synchronization barrier, nor while waiting for new tasks or locked surfaces to become available. However, Figure 5.1(b) shows an increase of total lock

idle time with the number of processors, and consequently an increase of the total execution time, explaining the obtained speed–up.



(a) Speed–up. (b) Cumulated times.

FIG. 5.1. *Performance results for the one floor test scene.*

Our individual lock instrumentation showed that the serialized access to the sorted list of surfaces represented 90% of the total lock time. Indeed, in this implementation, the processes first removed their receiving surface from the sorted list, then computed the radiosity transfer, before re–inserting the surface. In a huge model like a building, only a few surfaces are visible from a given emitting surface. Thus, all remaining non–visible surfaces were quickly removed and inserted in the sorted list, which became a bottleneck. Optimizing the sorted list management allowed us to enhance the parallel performance, as shown by the upper plot of speed–up of Figure 5.1(a). However, another lock (restricting access to the OpenInventor scene graph) emerged as a new bottleneck preventing to obtain a linear speed–up. This is a general remark, resolving a bottleneck often raise another one, often not expected as problematic, and this also points out the need of efficient management of critical data structures.

5.2. Practical experiment. To complete our evaluation, we planned to perform a radiosity computation for the entire Soda Hall building model and its furniture. However, the `arena` memory allocation library, and its parameters, made this quest difficult, since it appeared to allocate much more memory than necessary. Then, even with the 24 Gbytes of our machine, we were not able to have the computation completed. The key problem is that only 20% of the allocated memory is effectively used by our application.

Nevertheless, we succeeded to complete the computations for three floors of the model, all with furniture. This test model is composed of 320 113 surfaces, *i.e.* a little more than the five furnished floors performed in [9]. The final execution time, using 32 processors, was 11 hours, and about six millions of final mesh elements were created. Most of the lost time was spent on the different locks used by our implementation. Figure 6.1 shows a rendering of the sixth floor: note the fine shadows of the plants caught by the hierarchical radiosity meshing.

6. Conclusion. This paper presents an original system for computing highly precise radiosity solutions of large models on a conventional distributed shared memory multiprocessor machine. Solving the integral equation is done by a hierarchical algorithm, using high order wavelet basis functions, necessary to obtain both optimal accuracy and optimal finite element decomposition. Appropriate partitioning

and scheduling techniques are proposed to deliver optimal load balancing, by minimizing idle time waiting on locks and synchronization barriers for other tasks to be completed, while still exhibiting excellent data locality. The experiments we have performed show quite good parallel efficiency of more than 65% up to 32 processors.

Several practical implementation problems, mostly independent from the parallel algorithm itself, have arisen to limit the obtained parallel performance and associated scalability. First, we have found that a precise, efficient timing of idle time lost on locks is necessary to focus optimization efforts on the right places. Then, solving a bottleneck often creates another one, which in turn has to be resolved. However, we may expect this iterative tuning process to bring us to the optimal implementation. Finally, managing the large, dynamically growing memory necessary to achieve very large computations appears to be difficult but is a key issue to completing larger computations.

However, our experimentations allow us to expect a better scalability, both in terms of processor number and input database size. Our future work involves, among others, a theoretical study of the scalability properties of our parallel algorithm, which should confirm these experimentations. We believe that our approach, using distributed shared memory parallel computing, will be a more and more adequate way to answer the users' growing need — both in terms of large input dataset and execution time — as both small–scale multiprocessor personal workstations and larger scale supercomputers become available.

REFERENCES

[1] L. Alonso and N. Holzschuch, *Using graphics hardware to speed-up your visibility queries*, Journal of Graphics Tools, (1999). Submitted for publication. Also available as http://www.loria.fr/~holzschu/Publications/99-R-030.pdf.

[2] X. Cavin, *Load Balancing Analysis of a Parallel Hierarchical Algorithm on the Origin2000*, in Fifth European SGI/Cray MPP Workshop, Bologna, Italy, September 1999.

[3] X. Cavin, L. Alonso, and J.-C. Paul, *Parallel wavelet radiosity*, in Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation, Rennes, France, September 1998, Eurographics, pp. 61–75.

[4] A. Chalmers and E. Reinhard, *Parallel and Distributed Photo-Realistic Rendering*, ACM SIGGRAPH'98 Course Notes, July 1998. Course 3.

[5] M. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg, *A Progressive Refinement Approach to Fast Radiosity Image Generation*, in Computer Graphics (ACM SIGGRAPH '88 Proceedings), vol. 22, August 1988, pp. 75–84.

[6] M. Cohen, D. P. Greenberg, D. S. Immel, and P. J. Brock, *An Efficient Radiosity Approach for Realistic Image Synthesis*, IEEE Computer Graphics and Applications, 6 (1986), pp. 26–35.

[7] M. F. Cohen and J. R. Wallace, *Radiosity and Realistic Image Synthesis*, Academic Press Professional, Boston, MA, 1993.

[8] M. de Berg, *Linear Size Binary Space Partitions for Fat Objects*, in Proceedings of the 3rd Annual European Symposium on Algorithms, 1995, pp. 252–263.

[9] T. A. Funkhouser, *Coarse-Grained Parallelism for Hierarchical Radiosity Using Group Iterative Methods*, in Computer Graphics Proceedings, Annual Conference Series, 1996 (ACM SIGGRAPH '96 Proceedings), 1996, pp. 343–352.

[10] R. Garmann, *On the Partitionability of Hierarchical Radiosity*, Technical Report 702/1999, University of Dortmund, January 1999.

[11] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, *Modelling the Interaction of Light Between Diffuse Surfaces*, in Computer Graphics (ACM SIGGRAPH '84 Proceedings), vol. 18, July 1984, pp. 212–222.

[12] S. J. Gortler, P. Schroder, M. F. Cohen, and P. Hanrahan, *Wavelet Radiosity*, in Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings), 1993, pp. 221–230.

[13] P. Hanrahan, D. Salzman, and L. Aupperle, *A Rapid Hierarchical Radiosity Algorithm*, Computer Graphics (ACM SIGGRAPH '91 Proceedings), 25 (1991), pp. 197–206.
[14] P. Heckbert, *Simulating Global Illumination Using Adaptive Meshing*, ph.D. thesis, Technical Report, June 1991.
[15] J. T. Kajiya, *The Rendering Equation*, in Computer Graphics (ACM SIGGRAPH '86 Proceedings), vol. 20, August 1986, pp. 143–150.
[16] L. Renambot, B. Arnaldi, T. Priol, and X. Pueyo, *Towards efficient parallel radiosity for dsm-based parallel computers using virtual interfaces*, in Proceedings of the Third Parallel Rendering Symposium (PRS '97), Phoenix, AZ, October 1997, IEEE Computer Society, pp. 79–86.
[17] P. Schroder, S. J. Gortler, M. F. Cohen, and P. Hanrahan, *Wavelet Projections for Radiosity*, Computer Graphics Forum, 13 (1994), pp. 141–151.
[18] F. Sillion and C. Puech, *Radiosity and Global Illumination*, Morgan Kaufmann, San Francisco, CA, 1994.
[19] J. P. Singh, A. Gupta, and M. Levoy, *Parallel Visualization Algorithms: Performance and Architectural Implications*, IEEE Computer, 27 (1994), pp. 45–55.
[20] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, *Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity*, Journal of Parallel and Distributed Computing, 27 (1995), p. 118.
[21] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan, *Partitioning and Ordering Large Radiosity Computations*, in Computer Graphics Proceedings, Annual Conference Series, 1994 (ACM SIGGRAPH '94 Proceedings), 1994, pp. 443–450.
[22] R. Troutman and N. L. Max, *Radiosity Algorithms Using Higher Order Finite Element Methods*, in Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings), 1993, pp. 209–212.
[23] C. Winkler, *Expérimentation d'algorithmes de calcul de radiosité à base d'ondelettes*, PhD thesis, Institut National Polytechnique de Lorraine, 1998.
[24] H. R. Zatz, *Galerkin Radiosity: A Higher Order Solution Method for Global Illumination*, in Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings), 1993, pp. 213–220.

Acknowledgments. The authors would like to thank Alain Filbois, from the Centre Charles Hermite. Soda Hall model courtesy of Carlo H. Sequin.

Fig. 6.1. *Soda Hall sixth floor final image.*