Load Balancing Analysis of a Parallel Hierarchical Algorithm on the Origin2000

Xavier Cavin*
LORIA†- Équipe ISA,
Campus scientifique, BP 239, 54506 Vandœuvre-les-Nancy Cedex, France

Abstract

The ccNUMA architecture of the SGI Origin2000 has been shown to perform and scale for a wide range of scientific and engineering applications. This paper focuses on a well known computer graphics hierarchical algorithm - wavelet radiosity - whose parallelization is made challenging by its irregular, dynamic and unpredictable characteristics. Our previous experimentations, based on a naive parallelization, showed that the Origin2000 hierarchical memory structure was well suited to handle the natural data locality exhibited by this hierarchical algorithm. However, our crude load balancing strategy was clearly insufficient to benefit from the whole Origin2000 power. We present here a fine load balancing analysis and then propose several enhancements, namely "lazy copy" and "lure", that greatly reduce locks and synchronization barriers idle time. The new parallel algorithm is experimented on a 64 processors Origin2000. Even if in theory, a communication over-cost has been introduced, we show that data locality is still preserved. The final performance evaluation shows a quasi optimal behavior, at least until the 32*-processor scale. Hereafter, a problematic trouble spot has to be identified to explain the performance degradation observed at the* 64*-processor scale.*

1 Introduction

The emergence of many efficient hierarchical algorithms for solving very large numerical problems in scientific and engineering computing has brought out the need of dedicated parallel hierarchical architectures [5]. Since, several research projects have focused their work on the design, the implementation and the evaluation of hardware cache-coherent shared address space multiprocessors: among them, the MIT alewife machine [1], or the Stanford DASH prototype [14]. These studies have led to commercial realizations, such as the SGI Origin2000 [12], which is ccNUMA machine with a scalable distributed shared memory (DSM) architecture. The Origin2000 has been shown to deliver good performance by a recent evaluation paper [11], based on a wide range of kernels and applications from the SPLASH-2 suite.

Wavelet radiosity [10] is an efficient computer graphics method to compute the inter-reflections of light in Lambertian (*i.e.* diffuse) environments. The radiosity - power per unit area $[W/r^2]$ - on a given surface is defined as the light energy leaving the surface per unit area, and is governed by an integral equation involving all other surfaces of the input scene. This equation is solved using a finite element approach, which computes an approximation of the radiosity function as a finite linear sum of wavelet functions defined over $2D$ supports. The parallelization of wavelet radiosity is a key issue in order to bypass its huge computation time and memory requirements, and to compute simulations of complex environments in a reasonable time. However, as with all hierarchical N-Body methods, this algorithm has highly irregular and unpredictable data access patterns that make its parallelization challenging [16].

As suggested by [15] and demonstrated by a recent paper [9], distributed architectures, and their associated message passing paradigm, are definitely not well suited for parallel hierarchical radiosity algorithms. On the contrary, the shared address space, and the hierarchical memory structure, provided by the ccNUMA architecture appear to be particularly well adapted to handle their intensive communication and synchronization needs. The experiments we performed with our first naive parallelization [4] on the Origin2000 showed that an excellent data locality could be afforded with minimal efforts [3]. However, the crude load balancing implied by this naive parallelization did not allow us to get an optimal parallel performance. So now, what are the key problems involved in our simple load balancing strategy, and how can we solve them? Then, does the new parallel algorithm still present good data locality? And most of all, does it perform better?

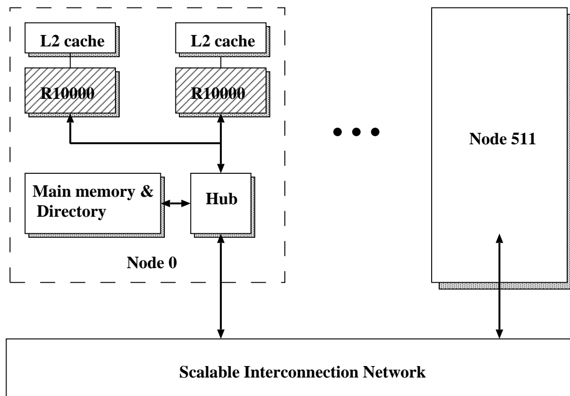
We start in Section 2 by briefly describing the ccNUMA architecture of the Origin2000, focusing on the main features affecting performance. In Section 3, we recall our previous work on the parallelization of the wavelet radiosity algorithm, and show how we could obtain a very good data locality. Then in Section 4, we highlight the load balancing problems that impede performance, and propose solutions reducing locks and synchronization barriers. In Section 5, we present and discuss the results of our enhanced algorithm, showing that despite the introduced communications and supplementary work, data locality is still preserved. We find that our new parallel algorithm can afford superlinear speed-up, at the small 16-processor scale, and acceptable speed-up at the moderate 32-processor scale. However, a trouble spot remains to be identified to explain the 64-processor scale performance degradation. Finally, conclusion and future work are presented in Section 6.

2 Understanding ccNUMA architecture

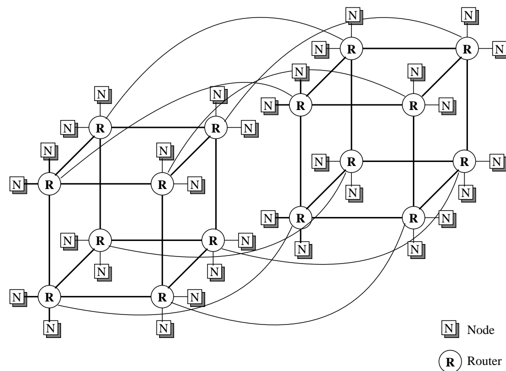
In order to fully benefit from a computer system performance, it is really important to understand the underlying architecture. As shown by Figure 1(a), the SGI Origin2000 is a scalable multiprocessor (up to 1024 processors) with distributed shared memory (DSM), based on the SN0 (Scalable Node 0) architecture [7]. The

*Xavier.Cavin@loria.fr
†LORIA is UMR 7503 LORIA, a joint research laboratory between CNRS, Institut National Polytechnique de Lorraine, INRIA, Université Henri Poincaré and Université Nancy 2.

basic building block is the node board, composed of two MIPS R10000 processors, each with separate 32 KB first level (L1) instruction and data caches on the chip with 32-byte cache line, and a unified (instruction and data), commonly 4 MB, two-way set associative second level (L2) off-chip cache with 128-byte cache line. Each node contains 64 MB to 4 GB of main memory (and associated directory memory), accessible through a custom circuit called the hub. Large SN0 systems are built by connecting the nodes together via a scalable interconnection network. Connecting two nodes is done by connecting their hub chips through a router, which can be itself connected up to six hubs or other routers. Figure 1(b) shows the topology of the 64 processors Origin2000 used for this paper: each node has 768 MB of main memory for a total of 24 GB.



(a) Block diagram.



(b) Topology of a 64 processors machine.

Figure 1: Scalable DSM Origin2000 [7].

The SN0 architecture allows the memory to be physically distributed, while making all memory equally accessible from a software point of view, in a ccNUMA approach. NUMA stands for non uniform memory access. Indeed, the time needed for accessing memory clearly depends on its location in the memory hierarchy (see Table 1). A given process only operates on data that are resident in its cache: as long as the memory is present in the cache, access times are very short; on the contrary, a delay occurs while a copy of the data is fetched from memory into the cache. The two processes of a given node have quick access through their hub to their local memory. Accessing remote memory through an additional hub adds an extra increment of time, as does each router the data must travel through. Moreover, as memory is manipulated through copies in the caches, it may happen that several processes have a cached copy of the same location at the same time. Thus, the first process modifying that data must instantly invalidate all other cached copies of the location, preventing other processes from using this "stale data". This is the issue cache coherence (cc), which is managed by the hardware in the SN0 architecture with a directory-based caches coherency scheme, as in [13].

Memory level	Read latency
L1 cache	5.1 ns
L2 cache	56.4 ns
Local memory	313 ns
Hub to hub (same router)	497 ns
Then for each hop	+100 ns

Table 1: Load latencies for the different memory levels [7].

Obviously, if the shared memory is seen as a contiguous range of virtual memory addresses, the physical memory is actually divided into pages, which are distributed all over the system. The default page size is of 16 KB, but it can be changed with the `dplace` utility. For *every* memory access, the operating system has to translate the virtual address into the physical address required by the hardware. A hardware cache mechanism, the translation lookaside buffer (TLB), keeps the 64 most recently used page addresses, allowing an instant virtual-to-physical translation for these 64 pages. This allows a 2 MB memory space (for the default page size) to be addressed without translation penalty. Programs using larger virtual memory may refer to a virtual address which is not cached in the TLB. In these case (TLB miss), the translation is done by the operating system, in the kernel mode: the memory read latency is then of approximately 10 000 ns, in the (common) case where the page has not been swapped to the disk.

As a conclusion, the SN0 architecture of the Origin2000 provides both the programming simplicity of a shared memory architecture and the scalability of a distributed memory architecture, by eliminating the finite bandwidth of a common bus. For our 64 processors machine, the number of router hops[1] is at the most of five and on average of 2.97, thus 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, that is the great majority of data accesses shall 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:

- *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;
- *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).

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

3 Early parallel wavelet radiosity

Our parallelization work takes place inside the CANDELA platform, which is a research project designed to provide a flexible architecture for testing and implementing new radiosity and radiance algorithms [17]. The CANDELA software is based on the SGI *Open Inventor* library, and consists of about 400 C++ classes.

We first briefly describe, as it is implemented inside CANDELA, the sequential algorithm we have chosen to parallelize. Then, we highlight the two key, but conflicting, aspects to face when parallelizing a hierarchical radiosity algorithm - load balancing and data locality - and we illustrate them, based on our sequential implementation. Finally, we present our two early parallel algorithms and the performance results we could obtain on the Origin2000.

3.1 Sequential algorithm

Our purpose here is not to enter into the underlying details of our sequential implementation, but rather to highlight the terms that will be important for our analysis. A more detailed presentation can be found in [4].

Basically, the sequential algorithm is based on the Southwell iterative method [6], and it proceeds as follows. A subset of the input scene surfaces have initial energy, either self-energy, or due to direct illumination by point light sources: they are first inserted into a *sorted list* of surfaces with energy to emit (or *residual energy*), sorted by decreasing energy. Then, successive *shooting iterations* are performed to update the scene surfaces radiosity function. At the beginning of each shooting iteration, the first surface, S_e, of the sorted list (*i.e.* the surface with the most residual energy) is removed from the list. Its residual energy is successively propagated to every other visible surface, S_r, of the scene: for each *interaction* between the *emitting surface* S_e and a *receiving surface* S_r, an *energy transfer* is computed to update the radiosity function of S_r. A part of the energy received by S_r is absorbed and the other is reflected, and thus added to the residual energy of S_r, whose position in the sorted list has to be updated consequently. A shooting iteration is completed after all energy transfers between S_e and its receiving surfaces S_r are completed. Finally, the residual energy of S_e is reseted, and the next shooting iteration begins, unless the desired convergence rate[2] has been reached.

In order to optimize the energy transfer for a given surface-surface interaction, we use 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. Figure 2 shows an interaction between a triangle and a parallelogram, and their associated hierarchical data structures. The energy transfer starts at the higher level of the quadtrees. An *oracle* function decides if the transfer can be done between the two current mesh elements, based on an evaluation of the committed error. If this error is too high, a lower level of one of the two surfaces (usually the biggest one) must be used: the energy transfer is recursively continued between the current mesh element of the non chosen surface and each of the lower-level mesh elements, that may have to be created, of the chosen one. Energy can thus be transfered between any levels of the quadtrees, as shown by Figure 2. 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.

[2] Ratio of total remaining residual energy over total initial residual energy.

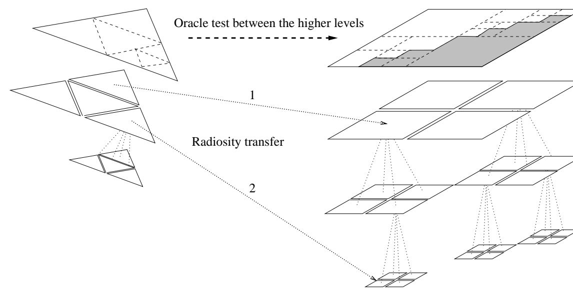


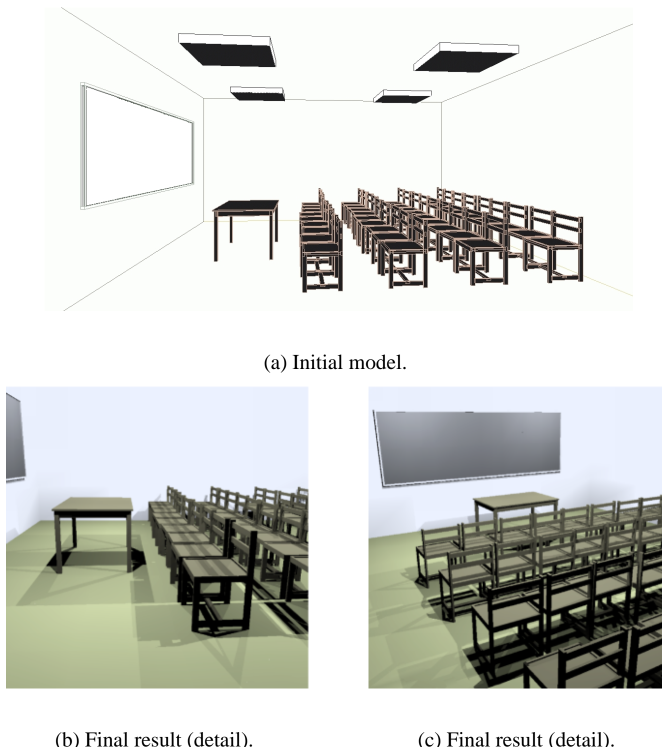
Figure 2: Hierarchical energy transfer.

The oracle function involves several user-defined parameters, that allow to control the number of mesh elements subdivisions. For instance, the limit size of a mesh element (`sizeLimit`), or the minimal ratio of brought energy over the already present energy (`radiosityRate`), under which no subdivision is done. Finally, the oracle function requires a great number of visibility computations to estimate the error due to the energy transfer: they are optimized using a binary space partition (BSP) of the scene, but could also be accelerated using available graphics hardware.

3.2 Overview of parallelization challenges

Load balancing

Let us consider the classroom model provided by Peter Shirley, which is a radiosity reference test scene. The initial model, showed on Figure 3(a), is composed of 3 153 initial surfaces and four emitting surfaces on the ceiling. Figures 3(b) and 3(c) show images of the radiosity computation result. The thin shadows on the floor show the precision of the solution, and thus the fine mesh elements decomposition done on the large initial floor surface.

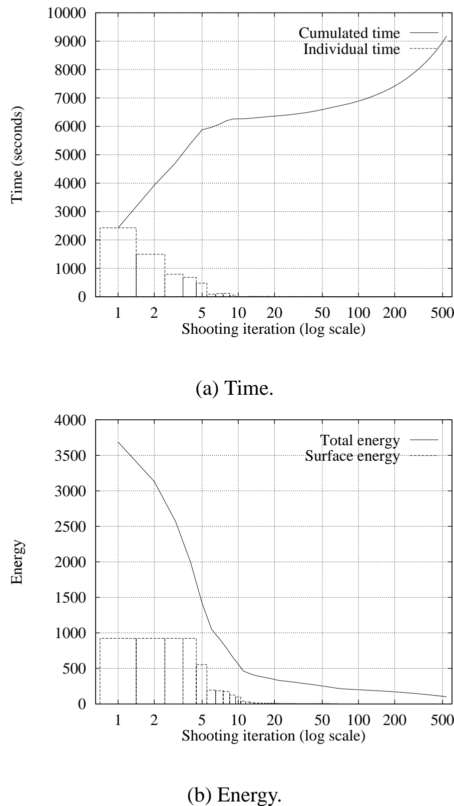


(a) Initial model.

(b) Final result (detail). (c) Final result (detail).

Figure 3: The classroom test scene.

The sequential execution took about 9 200 seconds to compute the 540 shooting iterations, for a remaining ratio of residual energy to shoot of 2 %. The final solution is composed of 62 218 mesh elements, including 47 448 final leafs. Figure 4(a) shows both the individual time needed to compute each shooting iteration (vertical boxes), and the cumulated time. The corresponding decrease of total residual energy is shown on Figure 4(b), where each vertical box represents the residual energy of the current emitting surface.



(a) Time.

(b) Energy.

Figure 4: Sequential experiment (classroom scene).

Although the classroom test scene is relatively simple, these two Figures illustrate the irregular and unpredictable characteristics of the wavelet radiosity algorithm. Indeed, the ten first shooting iterations are responsible for about 88 % of the total residual energy decrease and 68 % of the total execution time. The first four ones concern the four ceiling surfaces illumination, while the most important inter-reflections are handled by the next six ones. Then, the desired convergence is slowly reached with all the remaining short (between 4 and 10 seconds each), of low residual energy, ones.

As we can see on Figure 4, the computation time of a given shooting iteration is far from being proportional to the corresponding residual energy. Worst, a shooting iteration (for instance, the 8th) may take longer than another one (the 6th) having less residual energy. As we will see in the next Section, the same kind of irregularity can be found inside a shooting iteration.

Finally, the oracle function parameters may greatly influence the computation time of shooting iterations. For instance, the four ceiling surfaces have exactly the same initial residual energy, but their respective shooting iterations take less and less time! This is due to the `radiosityRate` parameter, here set to 5 %. Indeed, these shooting iterations concern the same receiving surfaces, which accumulate more and more energy. Thus, the energy transfers are done quicker on higher and higher levels of the quadtrees.

Data locality

As explained in Section 2, data locality is essential in order to sustain high parallel performance, especially on the Origin2000: it is necessary that the application achieves a very high ratio of cache hits at every level (L1, L2, TLB) of the memory hierarchy.

Let us consider another reference radiosity test scene, also provided by Peter Shirley, and shown on Figure 5. This scene consists of 10x10 dinner rooms, and contains 41 800 initial surfaces, among which 100 ceiling emitting surfaces. Large scenes like this one are problematic from a data locality point of view, for two main reasons. First, every new shooting iteration may involve totally different memory accesses from the previous one, depending on the location of the corresponding emitting surfaces in the scene. This is even more problematic with the dynamic creation of mesh elements. Indeed, the mesh elements of a given surface might have been created by different shooting iterations. Second, the visibility computations required by the oracle function may involve any surface of the scene.

As a consequence, any part of the constantly evolving memory may be accessed at any time during the whole computation.

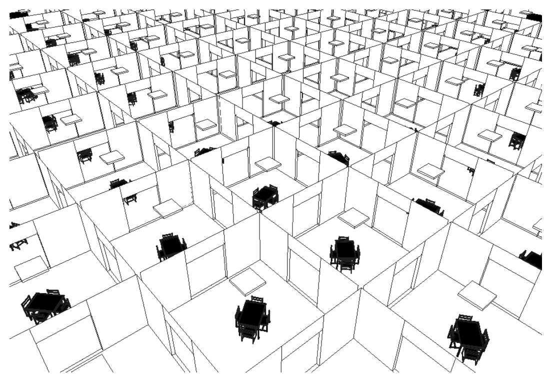


Figure 5: The dinner rooms test scene.

3.3 Parallel algorithms

An efficient approach to get an optimal data locality is to distribute the database among the different local memories, as in a message-passing scheme, thus restricting the processes to access a limited part of the whole memory. For instance, [2] uses a spatial subdivision of the input scene into sub-environments, which are distributed among processes: these ones only compute energy transfers for the surfaces they are assigned to, and energy exchanges between sub-environments are performed through *virtual interfaces*.

Unfortunately, if this kind of technique can be successfully applied to classical radiosity algorithms, it becomes rather inefficient for hierarchical radiosity ones, even with elaborated dynamic group partitioning methods [8]. Then, the alternate way is to exploit the natural data locality exhibited by these algorithms through the memory hierarchy of DSM computers, as suggested in [15].

Granularity choice

Our wavelet radiosity algorithm presents three levels of parallelism: across shooting iterations, across energy transfers (between input surfaces), and across mesh elements. Choosing the granularity for the parallelization is a difficult task, since it may greatly influence both load balancing and data locality.

The finest granularity (across mesh elements) allows an optimal load balancing, as in [16]. However, since a mesh element may be updated by several processes at the same time, it has to be locked for exclusive access: this approach has to be kept for radiosity simulations of small scenes, leading to a moderate number of mesh elements. We so have chosen to focus on the two remaining coarser granularities (across shooting iterations and across energy transfers) to implement the two following parallel algorithms.

ETL and SIL algorithms

Energy transfer level (ETL): this is historically the first parallel algorithm we have implemented. For simplicity reasons, we have chosen to perform the shooting iterations one after the other, in exactly the same order as in the sequential algorithm. For a given shooting iteration, all the energy transfers, for the corresponding emitting surface, are distributed for parallel processing. Each process starts by making an independent copy of the *complete* quadtree of the emitting surface, so that further mesh elements decompositions may be done on it without access conflicts. Then, it gets a receiving surface from a centralized list and computes the corresponding energy transfer. The copy of the emitting surface quadtree is reused for all assigned receiving surfaces, until no one remains in the list. At that point, the process waits on a synchronization barrier for all other processes to complete their energy transfer, before switching to the next shooting iteration. Synchronization barriers separating successive shooting iterations allow processes an exclusive access to the receiving surfaces, without any additional lock.

Shooting iteration level (SIL): in this algorithm, each process is now assigned to a complete shooting iteration. As in the ETL algorithm, a given process starts by making an independent copy of the quadtree of the emitting surface it has been assigned to, so that this surface is still able to receive energy from other emitting surfaces, as shown on Figure 7(a). Then, it successively computes the energy transfers for *all* its receiving surfaces, before it requests the next shooting iteration. Here, since many shooting iterations are performed at the same time, a surface is likely to simultaneously receive energy from several emitting surfaces, and so has to be locked for exclusive access.

First results

These two parallel algorithms have been implemented using IRIX `sproc`'ed processes. Since many mesh elements subdivisions occur at the same time during the computations, many memory allocations have to be done in parallel. We thus have overloaded the standard `malloc`, `free`, `realloc` and `calloc` functions, in order to provide a transparent, contention free, memory allocation package. Basically, an `arena` is allocated and assigned to each process, so that it can freely allocate its needed memory.

Our first experimentations [4, 3] on a 64 processors Origin2000 showed encouraging results, especially from a data locality point of view. Indeed, we could obtain cache hits rates of approximately 97 % for the L1 cache and 93 % for the L2 cache. This behavior can be explained by the two following kinds of data locality, as presented in Section 2. First, our application exhibits good spatial data locality thanks to the hierarchical quadtree data structures, combined with our memory allocation mechanism. Indeed, for a given surface, a large number of mesh elements decompositions are done by the same process, and are thus allocated on contiguous memory parts of the same `arena`. Second, the temporal data locality is also excellent, since, for a given energy transfer, both the quadtree data structures of the two interacting surfaces, and the BSP cells used for visibility computations, are reused many times. To summarize, the working set for a given energy transfer is definitely well adapted to the large caches of the Origin2000.

On the other side, load balancing, and consequently scalability, were not really satisfying. Indeed, we even did not obtain a linear speed-up at the small 16-processor scale, and the parallel efficiency at the moderate 32-processor scale was limited to about 50-60 %. These mitigated results are mainly due to our naive parallel algorithms, as we shall see in the next Section.

4 Enhancing load balancing

We first present in this Section a fine load balancing analysis of the implementation of the two parallel algorithms (ETL and SIL) presented in Section 3, highlighting both advantages and drawbacks in terms of scalability. Then, we propose two enhancements that partially resolved the raised problems. Finally, in order to benefit both from the advantages of the two algorithms, and from our introduced techniques, we present a brand-new parallel algorithm.

In order to illustrate our analysis, we will use tasks diagram representing the execution of our different algorithms, on the classroom test scene of Figure 3. For instance, Figure 6(a) shows parts of the tasks diagram for the sequential algorithm: each filled box represents a single energy transfer, its length being its computation time, and its color determining the emitting surface. A set of boxes of the same color corresponds to a complete shooting iteration, and thus to a single vertical box on Figure 4(a).

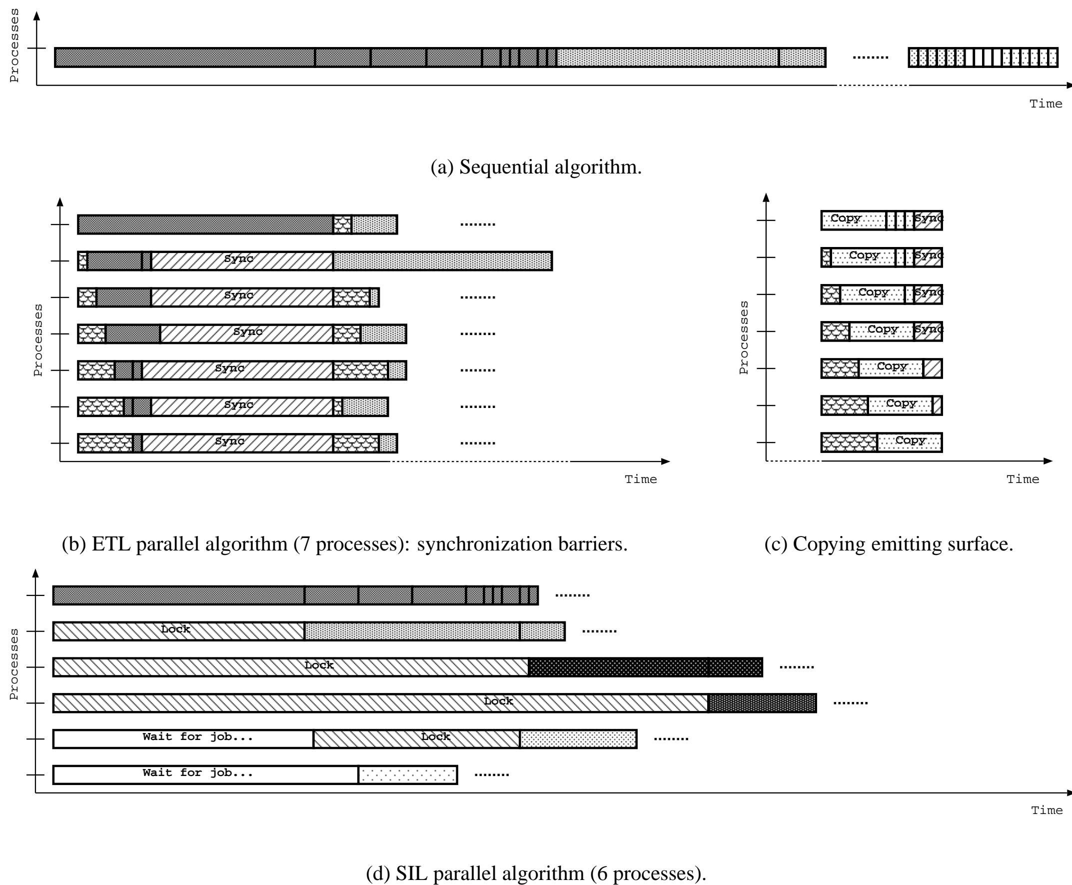
Tasks diagrams allow to give a better understanding of the behavior of our wavelet radiosity algorithm, especially at the energy transfer level. At the very beginning of the algorithm, emitting surfaces have a large residual energy (Figure 4(b)), and receiving surfaces are free of mesh elements subdivision. So, the first energy transfers are very irregular and unpredictable in term of computation times. On the classroom test scene (Figure 6(a)), the energy transfer between a ceiling emitting surface and the large floor will take many more time than with any other receiving surface, due to the many occluding chairs and the table. As the computation proceeds, emitting surfaces have less and less residual energy, and receiving surfaces accumulate more and more energy on their numerous mesh elements. Thus, the computation times of energy transfers become shorter and shorter, and more homogeneous.

Let us now see how these tasks can be distributed for parallel processing, while taking care for their preceding order. Indeed, some surfaces can only transfer their energy after having received it from other surfaces. On the contrary, for a given emitting surface, the order of receiving surfaces processing is not important.

4.1 Early algorithms analysis

Energy transfer level (ETL)

Parts of the tasks diagram for the ETL algorithm execution with seven processes on the classroom test scene are shown on Figure 6(b) and 6(c): hatched "Sync" boxes correspond to idle time waiting on the synchronization barrier between successive shooting iterations, and "Copy" boxes represent the time needed to make a copy of the emitting surface quadtree before the first energy transfer. The tiled boxes correspond to a side effect due to the use of the



(a) Sequential algorithm.

(b) ETL parallel algorithm (7 processes): synchronization barriers.

(c) Copying emitting surface.

(d) SIL parallel algorithm (6 processes).

Figure 6: Tasks diagram for the sequential and early parallel algorithms (classroom scene).

Open Inventor library: a short part right at the start of the copy has to be done inside a critical section, making processes begin their copy one after the other.

Synchronization barriers are particularly problematic at the beginning of the algorithm (Figure 6(b)). Indeed, inside a given shooting iteration, the energy transfers are likely to be very different in length, and are so quite impossible to be equally distributed. A few energy transfers are time consuming while others are shorter and quickly handled by remaining processes. Typically, on the classroom test scene, for each ceiling emitting surface, the first energy transfer to the floor surface may take more than half total shooting iteration computation time, due to the many shadows to catch. This is obviously more problematic as the number of processes increases: the more processes there are, the sooner the remaining of the energy transfers is completed, and the more time is lost on the synchronization barrier. For a given shooting iteration, if L_i $(i \in [1..T])$ are the respective computation times for the energy transfers to its T receiving surfaces, then the maximum speed-up is bounded by $\frac{\sum_{i=1}^{T} L_i}{\max_{i=1}^{T} L_i}$, and is independent of the number of processes. Fortunately, as long as the computation proceeds, energy transfers become shorter and can be better distributed, thus reducing synchronization idle time.

Copying the emitting surface quadtree is not problematic at the beginning of the algorithm. Indeed, surfaces are not yet fully decomposed into mesh elements, and copying is done quickly. On the contrary, the more the algorithm proceeds, the more surface quadtrees contain mesh elements, and the more duplication takes time. At the same time, energy transfer computation times become shorter. Then, copying the emitting quadtree may take longer than computing all the energy transfers! This is exactly what happens at the end of the computation on our classroom test scene, when the very subdivided floor has to emit a very few residual energy (Figure 6(c)): the first three processes complete quickly, but anyway slower than the sequential version, the short energy transfers, while the four remaining ones are still copying the emitting quadtree. In this case, using more processes would only grow the shooting iteration total execution time. Moreover, only the higher levels of the emitting quadtree are useful at the end of computation[3], so a huge part of the copy time is wasted (Figure 7(a)).

Shooting iteration level (SIL)

The SIL algorithm partially resolves the ETL algorithm problems, but unfortunately introduces new ones. Indeed, as in the ETL algorithm, the emitting surface quadtree for a given shooting iteration has to be duplicated, but the copy is only done once by the process that handles it, and is reused for all receiving surfaces. This allows to avoid the problems of Figure 6(c) due to the *Open Inventor* lock, but it still remains unsatisfactory when the duplication is longer than the computation of all the energy transfers.

Moreover, since several shooting iterations are handled in parallel, there is no need for synchronization barrier, except at the end, when the desired convergence has been reached. However, since any energy transfer may be performed at any time, care must be taken for accessing emitting and receiving surfaces. First, a given

[3] This is due to the oracle function: emitting surfaces have less residual energy while receiving surfaces have more accumulated energy.

surface can not receive several energy transfers at the same time, and so must be protected by a lock by the process that needs it. Second, when a process gets a new shooting iteration, the corresponding emitting surface may be locked as a receiving surface (previous case): it so has to wait for this energy transfer to be completed, then locks the surface to make an independent copy of its quadtree, and releases the lock, so that the surface can continue to receive energy transfers, as shown on Figure 7(a). Note here that duplicating the emitting surface quadtree is essential to avoid a *deadlock* between two processes computing energy transfers for two symmetrical interactions.

Figure 6(d) presents the tasks diagram for the beginning of the SIL algorithm execution on the classroom test scene, with six processors: hatched "Lock" boxes represent the time waiting on a locked surface for the associated energy transfer to be completed. We can see that this test scene is not particularly well adapted to this parallel algorithm. Indeed, only the four ceiling surfaces have residual energy at the beginning of the algorithm, so the four first processes deal with one ceiling surface shooting iteration each, and the remaining processes wait for new shooting iterations to be created ("Wait for job..." boxes).

The first energy transfer for each emitting ceiling surface shooting iteration concerns the floor surface: three processes have to wait for the remaining one, who got the lock first, to complete its computation. Once this first energy transfer is computed, the floor surface is unlocked, and another process can perform its energy transfer. Since the floor surface now have residual energy, a new shooting iteration has been created and assigned to a waiting process. This process, however, can not start its shooting iteration, since the floor surface is currently locked as a receiving surface. Obviously, this example illustrates the worst case, where many processes are waiting for the same surface involving time consuming energy transfers, but similar cases happen with standard, larger models. As the computation proceeds, energy transfer computation times decrease, but locking problems still remain, especially as the number of processes is growing.

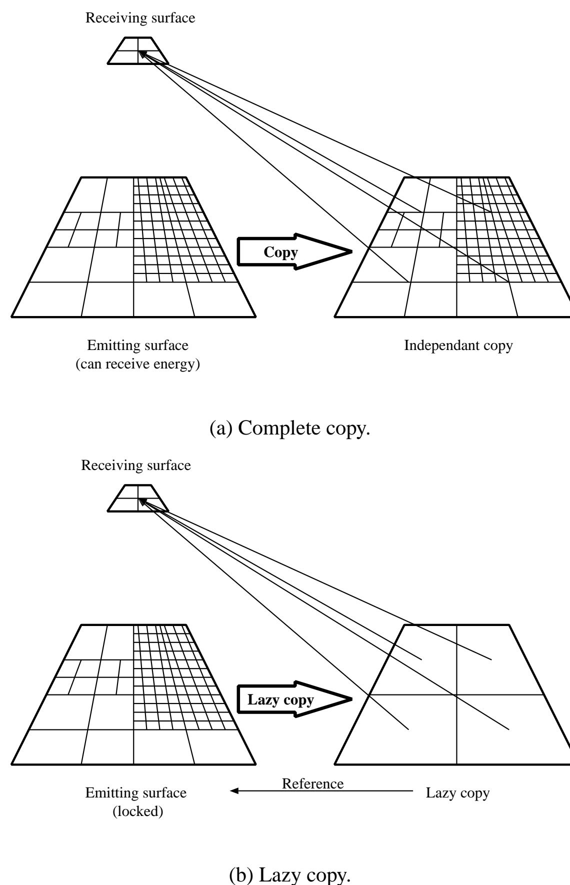
The last drawback of the SIL algorithm is that it converges slower than the sequential and the ETL algorithm, especially as the number of processes increases: supplementary shooting iterations may be required for the same result! For instance, on the classroom test scene, the first shooting iteration for the floor surface is performed with less residual energy as it would have if the four ceiling surfaces shooting iterations had been completed.

4.2 Lazy copy

Copying the emitting surface quadtree appears to be a critical point in the ETL and SIL algorithms. This is due to the following:

- the start of the copy must be done inside a critical section (not really problematic for the SIL algorithm);
- emitting surface quadtrees contain more and more mesh elements, thus increasing copy time, while shooting iteration computation times decrease;
- through the oracle function, only the higher levels of the duplicated quadtree remain useful as the computation proceeds.

The last two points, illustrated by Figure 7(a), naturally lead to the idea of using a light copy of the emitting surface quadtree. For building this so called "lazy copy", we no longer duplicate the whole quadtree, but only its higher level (the root), keeping a reference to the original quadtree. Then, when the process needs a lower level of the quadtree, it copies the information from the original quadtree, if it exists, or creates it on its lazy copy. Low levels of the quadtree are only copied when necessary, as shown by Figure 7(b). Further subdivisions done on the lazy copy are not copied back to the original quadtree, since they do not contain supplementary energy information.



(a) Complete copy.

(b) Lazy copy.

Figure 7: Energy transfer using a copy of emitting surface: in this example, only two levels of the duplicated quadtree are useful.

In order for that mechanism to work well, we have to ensure that the original surface quadtree will not receive energy while the shooting iteration is being computed. This is not problematic for the ETL algorithm, since the emitting surface will never be a receiving surface inside a given shooting iteration. Unfortunately, lazy copy can not be applied "as is" to the SIL algorithm, since locking the original quadtree would cause deadlocks. However, using lazy copy allows to dramatically reduce the ETL algorithm execution time, mostly at the end of the computation. Obviously, the more the emitting surface is subdivided, the more the gain is important, as with the floor surface of the classroom test scene.

4.3 Lures

The key problem appearing in the SIL algorithm is the restrictive access to the receiving surfaces (Figure 6(d)). Indeed, a lot of time may be wasted idling when many processes try to compute energy transfers towards the same receiving surface, especially when these are time consuming. This is the case for the floor of the classroom

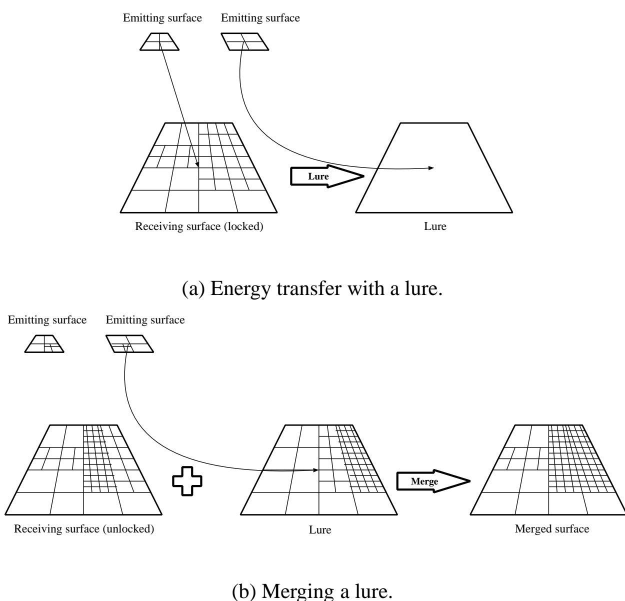
test scene, when it has to receive energy transfers from the four ceiling emitting surfaces.

We so had to find a mechanism so that a process no longer remains idle when it has to wait for a receiving surface to be unlocked: the basic idea is to have the process perform its energy transfer with a disjoint copy of the receiving surface, namely a *lure*. Once the energy transfer is completed, the computed lure quadtree and the original receiving surface quadtree must be merged.

Copying the receiving surface to build the lure could lead to the same problems as when copying an emitting surface. Indeed, duplicating the whole receiving surface quadtree may sometimes take longer than waiting for the other processes to complete their energy transfer. Fortunately, we do not here need the same kind of informations. We actually do not need the surface quadtree informations: only the "surface informations" are duplicated, and a brand new quadtree is build to compute the energy transfer (Figure 8(a)).

When a process has completed the energy 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, as done on Figure 8(b). In the other case, the process shall not wait for the surface to be unlocked, or all benefice would be lost. 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 can 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 currently having the lock, once it has completed its energy transfer, allowing the merging work to be divided among processes.



(a) Energy transfer with a lure.

(b) Merging a lure.

Figure 8: The lure mechanism.

To be complete, we have to note that the lure mechanism may introduce another supplementary work, depending on the oracle function parameters. Indeed, since a lure surface is considered with a new quadtree empty of energy, the `radiosityRate` parameter, taking into account the accumulated energy, can not play its role, as if the energy transfer was computed with the original surface.

4.4 A new parallel algorithm

On one hand, the ETL algorithm performs the same number of shooting iterations as the sequential algorithm and can be optimized with the lazy copy mechanism, but suffers from synchronization barriers idle time and can not be helped by the lure mechanism. On the other hand, the SIL algorithm does not suffer from synchronization barriers and may have locks idle time greatly reduced thanks to the lure mechanism, but it has to compute more shooting iterations for a same convergence and can not directly benefit from the lazy copy mechanism.

Then, the principle of our new parallel algorithm is to *continuously* process the energy transfers of the successive shooting iterations "in near the same order" as in the sequential algorithm. Actually, when all energy transfers of a given shooting iteration have been distributed, and even if some are not yet completed, the non-busy processes start computing a new shooting iteration, with 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 previous shooting iteration to be completed, but this avoids them to remain idle.

In order for that new scheme to perform efficiently, we have to combine the lazy copy and the lure techniques. Remember that for the lazy copy mechanism to work well, we have to ensure that the "lazy copied" surface quadtree is locked to prevent it from receiving energy transfers (Figure 7(b)). So, the solution to avoid a deadlock, when such an emitting surface has to receive an energy transfer from another shooting iteration, is to have it done on a lure.

Our new parallel algorithm is thus based on the energy transfer granularity, as the ETL algorithm. The tasks, consisting in a single energy transfer, are continuously assigned to processes, whose job is described by Algorithm 1.

Algorithm 1 Computing an energy transfer.

if *this is the first energy transfer of the shooting iteration* **then**
  - lock the emitting surface **(1)**
end if
- make a lazy copy of the emitting surface {/* since it is locked */}
if *the receiving surface is locked* {/* as emitting or receiving */} **then**
  - compute the energy transfer with a lure
  if *the receiving surface is still locked* **then**
    - add the computed lure to its list of pending lures
  else {/* the receiving surface has been unlocked */}
    - merge the computed lure quadtree (locking the receiving surface)
  end if
else {/* the receiving surface is not locked */}
  - lock the receiving surface
  - compute the energy transfer
  - merge the pending lure quadtrees, if any **(2)**
  - unlock the receiving surface
end if
if *this is the last energy transfer of the shooting iteration* **then**
  - merge the pending lure quadtrees, if any **(3)**
  - unlock the emitting surface
end if

The task of merging the quadtrees of the pending lures for a given surface is here assigned to the process in charge of unlocking this surface (Algorithm 1: **(2)** and **(3)**). During the merging operation, which may be time consuming, processes requiring this surface as a receiving surface for an energy transfer will still be able to use a lure. If it had been assigned to the process getting the first energy transfer for a shooting iteration with this surface as an

emitting surface (Algorithm 1: **(1)**), this would have delayed the following processes assigned to its next energy transfers.

To be complete, we have to note that a potential idle time problem remains, when a new shooting iteration starts, involving an emitting surface which is currently locked as a receiving surface (Algorithm 1: **(1)**). This could be avoided by introducing a *heuristic*, making a process using a lure when it has to compute an energy transfer towards a receiving surface which "is likely to become an emitting surface in a near future".

5 Results and discussion

5.1 Experimentation context

Our experiments were performed on the Origin2000 described in Section 2. The operating system running on it is the IRIX 6.5.4f release. The application was compiled with the MIPSpro 7.2.1 C++ compiler, since we still encounter severe compilation problems with the 7.3 release. We used the maximum performance optimization flags, selected by: `CC -n32 -Ofast=IP27`. Our program was instrumented to compute the idle time lost on our code *synchronization points* (locks and final synchronization barrier). We also used the performance analysis tool `perfex`, based on the R10000 hardware performance counters, to collect informations about the executions.

Let us now analyze the new parallel algorithm presented at the end of Section 4, applied on the two test scenes described in Section 3, and shown on Figures 3 and 5. For the dinner rooms test scene, we have chosen algorithm parameters allowing the whole memory (350 MB) to reside in the main memory of a single processor, in order to avoid artifactual sequential problems. For sake of completeness, the `sizeLimit` parameter was set to 0.1 meter, the `radiosityRate` to 0.01%, and the desired ratio of remaining total residual energy to 10 %: the sequential execution took about 30 hours to complete, creating 141 184 mesh elements.

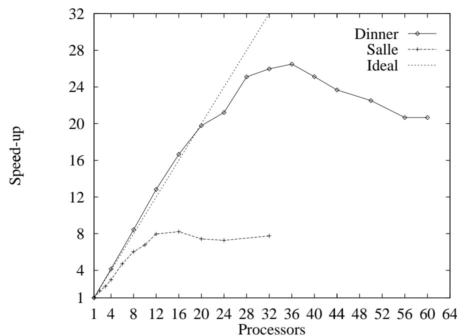
5.2 Results

Figure 9(a) shows the speed-up curves obtained on our two test scenes, with our new parallel algorithm. As expected, the classroom test scene does not exhibit enough concurrency, for our chosen granularity, to ensure parallel performance and scalability. On the contrary, the dinner rooms test scene appears to be particularly well suited to parallel computation, at least until 32 processors. A superlinear speed-up until 16 processors can even be underlined.

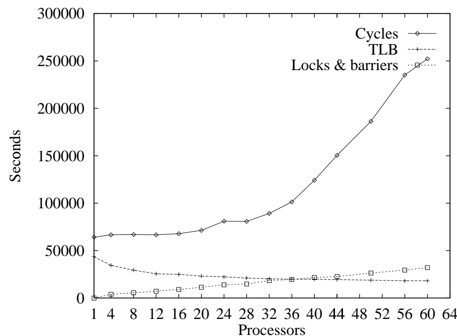
The L1 and L2 cache hits rates achieved with both scenes are very high, since they are respectively of 94% (L1) and 96% (L2), for any number of processors. However, these excellent results have to be mitigated by the TLB misses problem. Indeed, on the sequential algorithm, about 40% of the total execution time seems to be lost due to TLB misses. Figure 9(b) shows the evolution of the cycles (counter 0), the TLB misses (counter 23), as collected by the `perfex` tool, and the synchronization points idle time, as measured by our instrumented code, for the dinner rooms test scene. Finally, our application is not sensible to *false sharing*, as shown by the small values of counter 31 (about 10 seconds).

5.3 Discussion

The results obtained on the dinner rooms test scene, illustrated on Figure 9, are interesting, although confusing. On one hand, our hypotheses on data locality (Section 3) seem to be verified, since the L1 and L2 cache hits rate are very high, even if a problem appears at the TLB level. On the other hand, our new parallel algorithm allows an excellent load balancing, since the total idle time due to synchronization points slowly increases with the number of processors. Nevertheless, we observe three totally different phases for the obtained parallel performance.



(a) Speed-up on the classroom and dinner scenes.



(b) Execution analysis on the dinner scene.

Figure 9: Performance evaluation of our new algorithm.

At the small 16-processor scale, the number of cycles slowly increases from one to four processors, due to the parallelism overhead, and then nearly remains constant until 16 processors. At the same time, the number of TLB misses, which was very high with one processor, decreases from 43% with 16 processors. Thus, for a similar number of cycles, less time is spent in virtual addresses translation. Since, at the same time, idle time waiting on synchronization points does not increase, we obtain a superlinear speed-up. Then, at the moderate 32-processor scale, the TLB misses and synchronization points idle time curves are monotonous, but the number of cycles starts to "strangely" increase. However, this increase is light enough so that the speed-up remains acceptable (26 with 32 processors). Finally, at the large 64-processor scale, the strange increase of the number of cycles becomes so important that the speed-up is decreasing. We can also note that the synchronization points idle time curve is now over the TLB misses one.

A detailed analysis of the idle time lost on synchronization points shows that the idle time increase is not caused by load balancing problems. This is rather due to congestion happening when accessing critical data structures, such as the sorted list of emitting surfaces, which has to be constantly updated, or the tasks manager, distributing the energy transfers to the processes. The last point, for instance, could be solved by assigning a set of energy transfers to processes, and associate this with a *task stealing* mechanism.

From a data locality point of view, even if the L1 and L2 cache hits rates obtained by our application are very high, the main bottleneck appears to be the lack of TLB locality. However, this does not seem directly related to our parallel algorithm, but rather to operating system and/or hardware considerations. Indeed, using n processors allows to have $64n$ available TLB entries, thus greatly reducing the number of TLB misses.

The remaining trouble spot is the abnormal increase of cycles encountered at the 64-processor scale. With the state of our knowledges and existing tuning tools, we are currently not able to give a satisfying answer. We just can make some assumptions we would like to check in a near future. First, this may be due to the large number of TLB misses, since they are handled by an IRIX kernel routine. Second, our memory allocation package, based on IRIX `arena` involves many hidden locks protecting them: we plan to experiment the next version of this package, which completely removes these hidden locks.

6 Conclusion and future work

The new parallel wavelet radiosity algorithm introduced in this paper proves to deliver an optimal load balancing - by minimizing idle time waiting on locks and synchronization barriers for other tasks to be completed -, at least for large scenes exhibiting enough concurrency. Despite the communication over-cost introduced by our optimizing techniques, our application still exhibits excellent data locality when executed on the Origin2000, since it achieves very high L1 and L2 cache hits rates.

However, we discovered that our application suffers from lack of TLB locality, mostly due to the reduced number of TLB entries of the R10000 processor. An expedient to reduce TLB misses will be to use larger page sizes, thanks to the `dplace` command. This solution is suggested by SGI tuning guides, especially in the case of large dynamic database applications, what our program is not so far to be. Anyway, we showed that parallelism already allows to reduce TLB misses, by multiplying the number of TLB entries.

The 64-processor scalability of our application remains to be achieved. Indeed, we highlighted an important trouble spot, arising at the 32-processor scale, which is still unexplainable at the current state of our work. Moreover, load balancing may have to be enhanced, by optimizing parallel accesses to critical data structures, and maybe by reducing granularity to efficiently handle small scenes. Finally, it will also be interesting to test our application with much larger scenes, that is when the whole memory does not fit into the main memory of a single processor, in order to stress the Origin2000 memory system interactions.

Acknowledgments

We would like to thank the *Centre Charles Hermite* for providing us the computational resources, and critical access to its Origin2000. We are also grateful for the work of all the CANDELA project members, and especially Laurent Alonso, for his help and valuable discussions during the parallelization and tuning phases.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-J. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] B. Arnaldi, T. Priol, L. Renambot, and X. Pueyo. Visibility Masks for Solving Complex Radiosity Computations on Multiprocessors. In *Proc. First Eurographics Workshop on Parallel Graphics and Visualisation*, pages 219–232, Bristol, UK, September 1996.

[3] X. Cavin, L. Alonso, and J.-C. Paul. Experimentation of Data Locality Performance for a Parallel Hierarchical Algorithm on the Origin2000. In *Fourth European CRAY-SGI MPP Workshop*, Garching/Munich, Germany, September 1998.

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

[5] T. F. Chan. Hierarchical Algorithms and Architectures for Parallel Scientific Computing. In *Proceedings 1990 International Conference on Supercomputing, ACM SIGARCH Computer Architecture News*, volume 18, pages 318–329, Amsterdam, Netherlands, September 1990.

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

[7] D. Cortesi. Origin2000 (TM) and Onyx2 (TM) Performance Tuning and Optimization Guide. Tech Pubs Library guide Number 007-3430-002, Silicon Graphics, Inc., 1998.

[8] 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)*, pages 343–352, 1996.

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

[10] 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)*, pages 221–230, 1993.

[11] D. Jiang and J. P. Singh. A Methodology and an Evaluation of the SGI Origin2000. In *ACM Sigmectrics/Peformance*, Madison, Wisconsin, June 1998.

[12] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Denver, June 1997. ACM Press.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, June 1990. IEEE Computer Society Press.

[14] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–105, Gold Coast, Australia, May 1992. ACM Press.

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

[16] 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(2):118, June 1995.

[17] C. Winkler. *Expérimentation d'algorithmes de calcul de radiosité à base d'ondelettes*. PhD thesis, Institut National Polytechnique de Lorraine, 1998.