A Shared-Memory Implementation of the Hierarchical Radiosity Method

Axel Podehl

Fachbereich Informatik, Universität des Saarlandes, PF 151150, 66041 Saarbrücken, Germany, axelp@tibco.com

Thomas Rauber

Institut für Informatik, Universität Halle-Wittenberg, Kurt-Mothes-Str.1, 06120 Halle (Saale), Germany, rauber@informatik.uni–halle.de

Gudula Rünger

Fachbereich Informatik, Universität des Saarlandes, PF 151150, 66041 Saarbrücken, Germany, ruenger@cs.uni–sb.de

Abstract

The radiosity method is a simulation method from computer graphics to visualize the global illumination in scenes containing diffuse objects within an enclosure. A variety of realizations (including parallel approaches) were proposed to achieve a high efficiency while guaranteeing the same accuracy of the graphical representation. The hierarchical radiosity method reduces the computational costs considerably but results in a highly irregular algorithm which makes a parallel implementation more difficult. We investigate a task-oriented shared memory implementation and present optimizations with different behavior concerning locality and granularity. To be able to concentrate on load balancing and scalability issues, we use a shared-memory machine with uniform memory access time, the SB-PRAM.

Keywords Hierarchical radiosity method, shared memory implementation, task-parallelism, scalability, granularity.

1 Introduction

The radiosity method is a global illumination method from computer graphics to generate photo-realistic images of computer-generated three-dimensional environments with objects which diffusely reflect or emit light [16]. The method is based on the energy radiation between surfaces of objects and accounts for direct illumination and multiple reflections between surfaces within the environment. In contrast to the ray-tracing method (another popular global illumination method [15]) the radiosity method completely separates the shading and visible-surface determination. All interactions in an environment are first modeled in a view-independent stage, and then an image is computed using conventional visible-surface and interpolative shading algorithms.

The radiosity method decomposes the surface of objects in the scene into small elements with almost constant radiation energy. For each element, the radiation energy is represented by a *radiosity value*. The radiosity values of the elements are determined by solving a linear equation system relating the different radiation energies of the scene using configuration factors (which express the geometrical positioning of the elements). The computation of configuration factors as well as the solution of the linear system can be performed by different numerical methods (see [21] and its references).

A variety of methods have been proposed to reduce the computational costs, including an adaptive refinement technique [21], hierarchical methods [20], or progressive methods [8]. Adaptive and hierarchical methods reduce the number of configuration factors to be computed by combining the mutual dependencies of elements. The progressive method reduces the costs of solving the linear equation system. A further reduction of computation time is achieved by different parallel implementations [30].

The hierarchical radiosity method realizes an efficient computational technique for solving the transport equations that specify the radiosity values of surface patches in complex scenes [20]. The efficiency is achieved by computing the mutual illumination of surfaces more precisely for nearby surfaces and less precisely for distant surfaces. Similarly to other applications of the hierarchical approach (like the Barnes-Hut method [2] or the fast multipol method [17] for the solution of the N-body problem), the mutual influence decays with the square of the distance. So uniform accuracy is achieved with the hierarchical approach. The computation of fewer interactions results in a smaller system to be solved. Moreover, the system can be solved efficiently on the hierarchical data structure supporting the hierarchical method.

In this article, we investigate the implementation of the hierarchical radiosity method on shared memory machines. We use a task-oriented shared memory programming model which is well-suited for the parallel implementation of irregular problems like the hierarchical radiosity method. The irregularity comprises irregular, dynamic data structures and fine-grained shared data accesses. The starting point of the investigation is the implementation from the SPLASH-2 application program suite [38] which was designed to facilitate the study of centralized and distributed shared-address-space machines. The detailed description in [24,30,31] shows that the SPLASH-2 implementation of the hierarchical radiosity method is tuned for an execution on a cache-based virtual-shared-memory machine with a physically distributed memory (Stanford DASH) [22]. Load balance is realized by using distributed task queues with task stealing. This interferes with the locality of accesses to the task queues, because each attempt to steal a task includes an access to a remote task queue.

An important question in parallel computing is the scalability of parallel applications, which describes how the speedup of the parallel algorithms (expressed with a parametric number of processors) depends on an increasing number of processors. A requirement for a good scalability is that the potential degree of parallelism exceeds the number of processors so that all processors can be employed and that the granularity of tasks enables a good load balance. Usually, the competition between load balance (or granularity) and data locality hinders a concise study of scalability. To overcome this difficulty, we use the SB-PRAM which provides a large number of processors and a global shared memory [1]. Because the SB-PRAM has uniform access time, the implementation can concentrate on the efficient exploitation of task granularity and can ignore issues of locality. The original implementation is optimized on the algorithmic level, on the design level for tasks (to achieve a finer granularity), and on the task administration level.

We present the parallel programs in a task-oriented shared memory model where tasks can initiate new other tasks to be performed on different processors. The parallel programs investigated represent realizations of the general specification of the radiosity algorithm defining the maximum degree of parallelism. Both implementations show good speedup values on the SB-PRAM for small number of processors, exceeding the speedup values on the DASH. But only the optimized version also exhibits good speedup values for large numbers of processors (up to 2048).

The remainder of the paper is organized as follows. Section 2 summarizes the standard and the hierarchical radiosity methods. Section 3 presents the shared memory implementation. Section 4 describes the numerical experiments. Section 5 discusses related work and Section 6 concludes.

2 Radiosity methods

The radiosity method simulates the global illumination in scenes containing diffuse, opaque objects within an enclosure. The radiation from surfaces of objects not emitting radiant energy themselves is caused by illumination by radiant objects. The general radiosity method follows the simulation of thermal heat transfer as, e.g., described in [28]. Principal assumptions include the diffuse emission and reflection of surfaces, a non-participating medium, and a uniform intensity within each zone (a zone is an indivisible unit of a surface). Further assumptions are wavelength-independence and constant radiation in each zone. The latter assumptions do not restrict the application area. Wavelength-independence can be achieved by considering all wavelengths independently; constant radiation in each zone can be achieved by subdividing the zones into smaller ones. In the following subsections, we summarize the standard radiosity method and the hierarchical version.

2.1 The standard radiosity method

The radiosity method starts by subdividing the surfaces of the objects in the scene into a number of small pieces A_j, $j = 1, \dots, n$, which are called *elements*. For each of the elements, a *radiosity value* B_j (of dimension $[\mathrm{Watt/m^2}]$) is computed which describes the specific radiant energy per unit time and per unit area dA_j of A_j. The radiosity values of different elements are not independent but influence each other. We briefly outline the derivation of the linear system specifying the dependencies between radiosity values.

The radiance $\Phi_j = B_j A_j$ (of dimension $[\mathrm{Watt}]$) of an element A_j into a specific direction (described by the angle δ_j between this direction and the normal vector of the element A_j) and a specific region (described by the solid angle Ω intervening the region) is given by $\Phi_j = I(\delta_j)\Omega$. The surfaces are assumed to be *Lambertian* reflectors (also called diffuse objects) which scatter light equally in all directions due to the Lambertian cosines law, $I(\delta_j) = L_j A_j \cos\delta_j$, i.e., the intensity I_j depends linearly on the reduced radiant surface $A_j cos\delta_j$ corresponding to the inclination in direction δ_j with a constant radiant density L_j. The entire radiant power of A_j is calculated by integrating the radiant power per unit solid angle $d\Omega$ in the hemisphere illuminating the space. This results in the equality for the specific radiosity:

$$B_j = \frac{\Phi_j}{A_j} = \int_{\Omega} L_j \cos\delta_j d\Omega = L_j\pi \tag{1}$$

Because of the mutual illumination of diffuse objects, the radiosity value B_j consists of the elements's emission energy E_j and radiosity values $H_i = B_i F_{ij}$ which are portions of the radiosity values B_i of other elements A_i not occluded by intervening elements. Thus, a specification of all radiant energies $\Phi_j =$

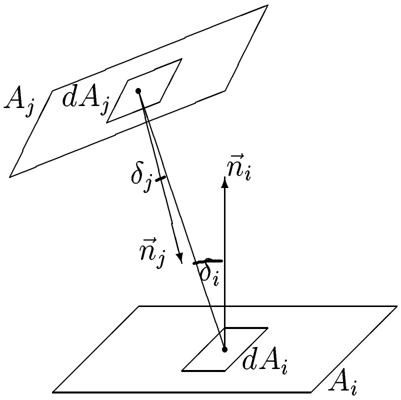

Fig. 1. Energy transport between differential areas dA_i and dA_j.

B_jA_j, $j = 1, \ldots, n$, has the form of a linear equation system

$$B_jA_j = E_jA_j + \rho_j \sum_{i=1}^{n} F_{ij}B_iA_i, \quad j = 1, \ldots, n. \tag{2}$$

The dimensionless factors F_{ij} (called *configuration factors* or *form factors*) describe the portions of Φ_i incident on A_j. The configuration factors depend only on the geometrical positioning of the two elements A_i and A_j. The factor ρ_j describes the diffuse reflectivity property of A_j.

For diffuse objects the configuration factors can be described more precisely. The factor $F_{dA_i-dA_j}$ describing the portion of radiant energy Φ_i of an infinitesimal area dA_i of A_i incident on an infinitesimal area dA_j of A_j corresponds to the portion of the radiant energy $d\Phi_i$ of dA_i in the direction of the solid angle $\Omega_j = \frac{dA_j\ cos(\delta_j)}{r^2}$ subtended by dA_j (where δ_j is the angle between the normal vector of A_j and the direction into A_i, and r is the distance between the elements, see Figure 1). The factor $F_{dA_i-dA_j}$ is the quotient of $d\Phi_i(\Omega_j) = L_i\ dA_i\ cos(\delta_i)\ \Omega_j$ (the energy from dA_i incident on dA_j according the Lambertian law) and the total radiation $d\Phi_i = dA_i\ L_i\pi$ of dA_i (according to Equation (1)), i.e.,

$$F_{dA_i-dA_j} = \frac{d\Phi_i(\Omega_j)}{d\Phi_i} = \frac{cos(\delta_i)cos(\delta_j)dA_j}{\pi r^2},$$

The fraction of lighting from the whole element A_i incident onto the whole element A_j results from integrating over the elements dA_j and taking the mean value of the infinitesimal elements dA_i:

$$F_{A_i-A_j} = F_{ij} = \frac{1}{A_i} \int\limits_{A_i} \int\limits_{A_j} \frac{cos(\delta_i)cos(\delta_j)}{\pi r^2} dA_j dA_i. \tag{3}$$

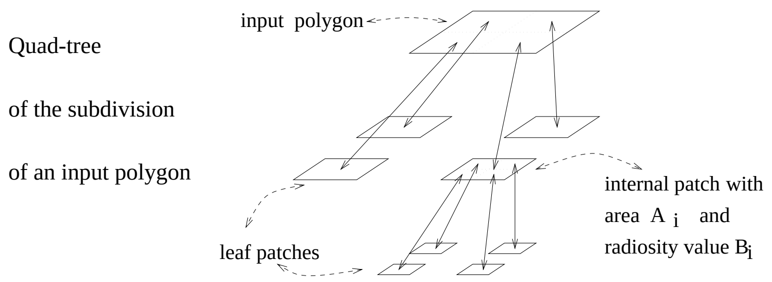

Fig. 2. Quadtree representing a subdivision into patches.

The configuration factors are symmetric in the sense $F_{ij}A_i = F_{ji}A_j$ and for one area A_i the sum of all fractions of the radiosity in an enclosure describes the entire radiosity, i.e., $\sum_{j=1}^{n} F_{ij} = 1$. Using the symmetry relation, Equation (2) yields the linear system of equations for the radiosity values B_j (see [11,26]):

$$B_j = E_j + \rho_j \sum_{i=1}^{n} F_{ji} B_i, \quad j = 1, \ldots, n. \tag{4}$$

The radiosity method first computes the $n(n-1)/2$ configuration factors according to Equation (3) and then solves the linear system (4) with an appropriate solver for linear equation systems as the Jacobi and the Gauss-Seidel method or an alternative iteration scheme like the *shooting* of radiosity values [8]. The asymptotic runtime is $O(n^2)$ for computing the configuration factors and solving the system. In order to reduce the runtime, the hierarchical version of the radiosity method reduces the number of interacting elements.

2.2 The hierarchical radiosity method

The most expensive parts of the radiosity method are the computations of the configuration factors between different elements. The *hierarchical* radiosity method [20] reduces the $O(n^2)$ time complexity for computing the configuration factors to $O(n + k^2)$ (where k is the number of input polygons representing the surface of objects) by adaptively subdividing polygons into a hierarchy of smaller pieces of the surfaces. Depending on the energy exchange between polygons, form factors (and thus the energy exchange) are computed on different levels of the subdivision but with similar accuracy.

For each input polygon the subdivision into the hierarchy is organized in a quadtree, see Figure 2. The inner nodes of the quadtree are called *patches* and the smallest pieces attached to the leaves are called *elements*. The patches or elements attached to the four children of a node represent a partition of the patch attached to the parents node. The division of patches is stopped depending on a stopping criterion using A_ϵ (with an element having minimal size $A_\epsilon/4$). The set of elements attached to the leaves of the quadtree represents

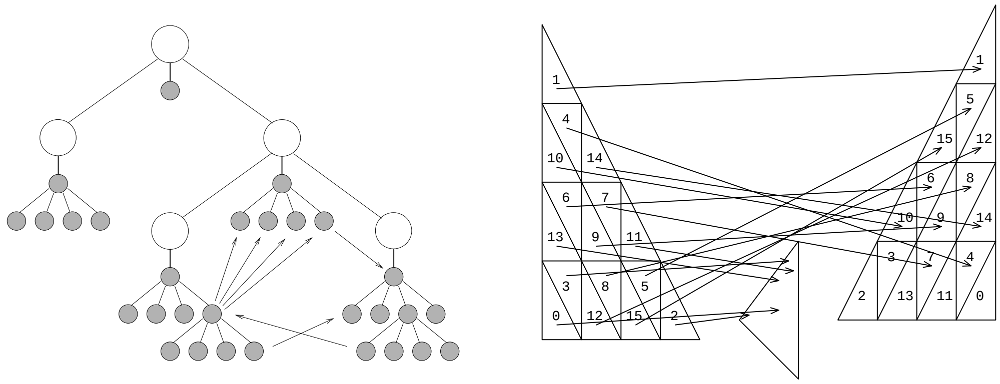

Fig. 3. (a) Mixed BSP quadtree data structure and (b) Visibility test ray tracing.

the partition of the polygon into elements. For each patch q of each input polygon, an interaction list is maintained containing patches of other input polygons, which are visible from the patch q, i.e., which can reflect light to q. The interaction lists are constructed during the construction of the quadtrees by recursive subdivision.

The input polygons themselves are organized in a BSP tree (binary space partitioning tree) which is used for the computation of visibility between patches. The BSP tree is constructed by recursively selecting a root polygon R of the (sub)tree and partitioning the associated polygons in two sets: one set contains the visible polygons lying on the reflecting side of R, the other contains the polygons that are not visible for R. A polygon lying on both sides is subdivided accordingly. For a fast visibility computation, the root polygon must be chosen such that the two sets contain about an equal number of input polygons, leading to a balanced BSP tree. For further details see [29]. Each node of the BSP tree represents the root of the corresponding quadtree of the hierarchical subdivision of the particular polygon. The entire data structure is illustrated in Figure 3 (left). The union of the elements attached to leaves of all quadtrees of this data structure represent the elements A_i, $i = 1, \ldots, n$, of the scene.

Figure 4 presents the hierarchical radiosity method in pseudo-code notation. The algorithm starts by building up the BSP tree in the initialization step (see phase (a) in Figure 4). During the computation of the form factors, the quadtrees are built up adaptively in order to guarantee the computations to be of sufficient precision. Therefore, the quadtrees need not be balanced. The method computes the energy transport (i.e., the configuration factor) between two patches or elements only if it is not too large; otherwise the patches are subdivided. Thus, each patch or element has its individual set of interaction elements or patches for which the configuration factors have to be computed.

For each patch, the decision about the set of interacting elements depends on an a-priori estimation of the influence of the radiosity. A purely geometrical approach considers the form factor $F_{A_i-A_j}$ between a radiant element A_i and an illuminated element A_j and approximates it by the relation of the distance r of two patches and the expansion of the illuminated element. Configuration factors are computed only between elements of appropriate energy transport which must not be larger than a predefined error bound F_ϵ. The error bound guarantees that the distance r between the patches is large enough compared to the area A_j of the illuminated patch and, thus, the formula for the configuration factor is valid.

A-priori-estimations mainly based on the geometrical relationship do not take into account the different degrees of radiosities which influence the energy transport. The geometrical approach can be improved when approximations of the radiosity values B_i are known. The hierarchical method in Figure 4 alternates iteration steps of the Jacobi method to solve the energy system (4) with a re-computation of the quadtree and the interaction sets based on $F_{ji}B_i$ with radiosity values B_i from the last iteration step. The values $F_{ji}B_i$ are estimated and must not be larger than a bound B_ϵ. This approach is called the hierarchical radiosity method with *radiosity based subdivision*. The iteration stops when the difference of the total radiosity of two successive iterations is small enough.

There is a variety of methods to estimate the form factors, to compute the visibility factors, or to perform iteration steps solving the linear system. In this paper, we consider the version of the radiosity method described in [20] and adopted in the implementation of the SPLASH-2 benchmark suite [38]. The computations (marked by the same numbers (i)-(iv) in Figure 4) are realized in the following way.

(i) Estimation of form factors: The form factors are approximated numerically due to the difficulty of solving the integrals (3) in closed form. One possibility is a ray tracing method proposed in [37]. The form factor from an area A_i to an area A_j is approximated by a form factor from four points of A_i (which are points on disjoint subareas A'_i of A_i) to four small disks A_{D_l}, $l = 1, \ldots, 4$, covering A_j. This form factor is computed by

$$F_{A'_i-A_{D_l}} = \frac{cos(\delta_i)cos(\delta_j)\pi R^2}{\pi r^2 + \pi R^2}$$

where R is the radius of the disks A_{D_l}. The sum of the form factors of the four disks covering A_j is taken as the approximation of the form factor $F_{A'_i-A_j}$ from a point on A_i to the complete area of A_j. For the computation of $F_{A_i-A_j}$, the mean value of the point-to-area form factors $F_{A'_i-A_j}$ is taken. The accuracy of the approximation can be changed by considering different coverings of A_j or different numbers of points on the radiant surface A_i.

(a) Initialization:

Insert all input polygons R into a BSP tree;
Attach the root of a quadtree to each node of the BSP tree;
Initialize interaction sets $I(R)$ of roots of quadtrees with visible input polygons;
Initialize radiosity values of input polygons with $B_R = E_R$, for all polygons R;

(b) Radiosity-based division:

For each polygon R in the BSP tree
  For each patch j in the quadtree for R (top-down)
    For each interaction $i \in I(j)$, ($I(j)$= interaction set of j)
      Compute configuration factor F_{ji}; (i)
      Compute visibility factor V_{ji}; (ii)
      if ($V_{ji}F_{ji}B_i > B_\epsilon$ and $A_i, A_j > A_\epsilon$) then Divide (A_j, A_i); (iii)

(c) Iteration step:

For each polygon R in the BSP tree
  For each patch j in the quadtree for R (top-down)
    Compute the radiosities B_j^{new} according to Equation (5) (iv)
  For each patch j in the quadtree for q (bottom-up)
    Compute the radiosities B_j^{new} according to Equation (6) (iv)

(d) Error control:

Stop if the error is small enough; otherwise repeat step (b);

(e) Bilinear interpolation for edges of the elements to smooth the solution;

Fig. 4. The hierarchical radiosity method with radiosity based subdivision.

(ii) Computation of visibility factors: The radiosity between patches or elements can be decreased by intervening occluding elements. The factor $V_{ij} \in [0, 1]$ corrects the form factor from A_i to A_j to $V_{ij}F_{ij}$. If $V_{ij} = 1$ then the two patches are completely visible to each other; if $V_{ij} = 0$ then the patches are completely occluded. The visibility test proposed in [20] uses a ray tracing method with a fixed number of rays between two patches. Here we use rays from 16 sub-elements to 16 sub-elements (see Figure 3, right). The percentage of rays not blocked by intervening surfaces is used as visibility factor. The BSP tree is used to reduce the computational effort of the visibility test: polygons potentially occluding two other patches have to be in front of the surfaces and are located in the BSP tree appropriately.

(iii) Subdivision step: The subdivision of patches is performed if the energy transport between two patches is not small enough, i.e. $V_{ji}F_{ji}B_i > B_\epsilon$ with a predefined error bound B_ϵ. If the surface A_j is larger (i.e., $A_j > A_i$) then this area A_j is divided and the interaction lists are adapted appropriately: the patch A_i is deleted from the interaction list of A_j and is inserted into the interaction lists of the children of A_j. If $A_j < A_i$ then A_i is divided:

A_i is deleted from the interaction list of A_j, and the children of A_i are inserted into the interaction list of A_j.

(iv) Iteration step: An iteration step of the Jacobi method for solving System (4)

$$B_j^{(l+1)} = E_j + \rho_j \sum_{i=1}^{n} V_{ji} F_{ji} B_i^{(l)}, \quad j = 1, \ldots, n, \quad l = 0, 1, \ldots,$$

mainly consists of a matrix-vector multiplication of the vector of radiosity values with a generalized block-matrix $(V_{ji}F_{ji})_{i,j=1,..,n}$ consisting of rectangular blocks with identical entries. Blocks with more than one entry arise when energy exchanges are computed on a higher level than the element level and, thus, all elements in that subtree have the same average form factor, see Figure 5. The matrix-vector multiplication is performed on the quadtree structure in the following way, see [29] (again, the patches are denoted by j, the interacting partner in the interaction set $I(j)$ are denoted by q): The quadtrees for the different input polygons are traversed top-down and for each patch j, a radiosity value is computed by considering the interaction with all patches in the interaction list $I(j)$ of j. If patch j is not the root of a quadtree, the radiosity value of j's parent node is propagated to j. In particular, the following radiosity value is computed:

$$B_j = \begin{cases} \rho_j \sum_{q \in I(j)} F_{jq} B_q & \text{for a root patch } j \\ \rho_j \sum_{q \in I(j)} F_{jq} B_q + B_{\text{parent}(j)} & \text{for an inner nodes } j \\ \rho_j \sum_{q \in I(j)} F_{jq} B_q + B_{\text{parent}(j)} + E_j & \text{for a leaf } j \end{cases} \tag{5}$$

After the radiosity values for all patches are computed, a bottom-up traversal of the quadtrees is performed to accumulate the area-weighted radiosities of a patch's descendants into its own radiosity. If a patch is always subdivided into four sub-patches of equal size, the radiosity values are updated in the bottom-up traversal in the following way:

$$B_j = \frac{1}{4} \sum_{r \in children(j)} B_r \tag{6}$$

If the radiosity values of the scene have not yet converged, the next iteration performs the similar traversals of the quadtrees using the radiosity values of the previous iteration as input.

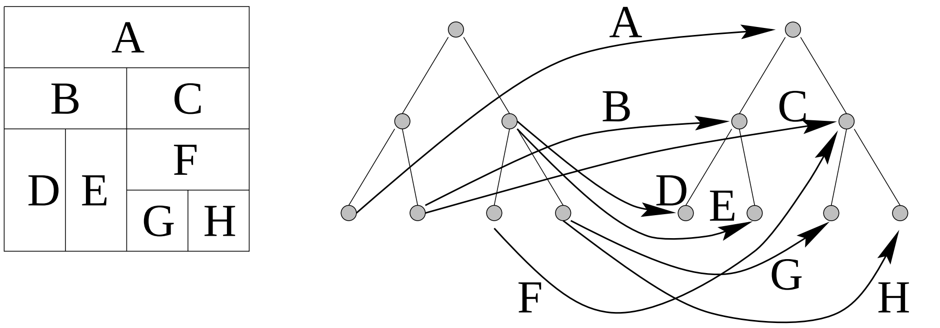

Fig. 5. Example for a block-constant matrix and corresponding quadtrees with interactions for the case of two interacting input polygons.

3 Parallel implementation of the hierarchical radiosity method

In this section, we describe two shared memory implementations of the hierarchical radiosity method with radiosity based subdivision. For the parallelization, we have chosen a task-oriented shared memory model that is well-suited for the implementation of irregular problems. A parallel implementation according to this programming model is based on an organization of the algorithms into a set of tasks; the set of tasks determines which parts of the potential parallelism can actually be exploited. Figure 6 shows a pseudo-code program of the hierarchical radiosity method which expresses the potential parallelism explicitly. Portions of the program that can be executed independently from each other in parallel are separated by a $||$ symbol. Loops with independent iterations are described by <u>forall</u> symbols. Recursive parts of the algorithm are expressed by <u>do recursively</u> constructs.

Sorting the input polygons $R_1, \ldots R_k$ such that the insertion in the initially empty BSP tree results in a balanced tree must be executed sequentially because the selection of polygon R_i depends on the selection of the previous polygons, see [35]. After the sorting, the polygons are inserted one after another in the BSP tree and the initial form factors and visibility factors are computed, which can be done in parallel. If the visibility factor V_{R_j,R_i} between two polygons R_j and R_i is not zero, R_j is inserted into the interaction list of R_i and vice versa. Otherwise, no interaction can occur and nothing needs to be done. After the initialization, the recursive subdivision and interaction computation can be executed for each polygon in parallel. All interactions of each polygon can also be computed in parallel; this is realized in the procedure $\mathsf{compute\text{-}interaction}()$. In the radiosity-based subdivision, the procedure $\mathsf{Divide}(A_j, A_i)$ also includes updates of the interaction lists.

The hierarchical radiosity method exhibits several sources of irregularity. In each iteration step, the interactions between patches or elements are reconsidered which may result in a change of the interaction levels or in new subdivisions. This leads to an irregular change of interaction relations, i.e., the levels

```
(0) Prephase: Sort input polygons R_1, ... R_k to balance BSP tree;
(1) Phase 1:
      do recursively: insert-polygon(R_1)
      with procedure insert-polygon(R_i) = {
         Insert R_i into BSP tree;
         do in parallel {
            insert-polygon(R_{i+1})
            || forall R_j with j < i do {
                  Compute F_{R_j,R_i} and F_{R_i,R_j}
                  || { Compute visibility factors V_{R_j,R_i};
                       if (V_{R_j,R_i} ≠ 0)
                          Insert R_j in interaction list of R_i and vice versa; }
               }
         }
      }
(2) Phase 2:
      do {
         forall polygons R ∈ {R_1, ..., R_k}
            do recursively: compute-interaction(root(R));
         with procedure compute-interaction(j) = {
            forall interactions i ∈ I(j) do {
                  Compute visibility factor V_ji;
                  if ( V_ji F_ji B_i > B_ε and A_j, A_i > A_ε ) then {
                     Divide(A_j, A_i);
                     compute configuration factors of new interactions;
                  }
            }
            Compute B_j = ρ_j Σ_{i∈I(j)} V_ji F_ji B_i + B_{parent(j)};
            if j is leaf then {
               B_j = B_j + E_j;
               while (radiosity values of all children of
                                   i have been computed) do {
                  j = parent(j);
                  B_j = 1/4 Σ_{i=child(j)} B_i;
               }
            }
            else { forall children r of j do compute-interaction(r) ;}
         }
      } while (ERROR > ε)
(3) Phase 3:
      forall elements in all quadtrees do
         { bilinear interpolation for edges of the elements; }
```

Fig. 6. Maximum degree of parallelism of the hierarchical radiosity method.

and the number of interactions and the number of radiosity values may vary irregularly during the computation of the scene. Moreover, the BSP tree of input elements might be unbalanced, different quadtrees may contain different numbers of elements, and each single quadtree can be unbalanced. A more balanced BSP tree can be achieved by using heuristic algorithms for sorting the input polygons before inserting them (sequentially) into the BSP tree [35].

We now present the programming model and the specific execution platform which we use for the implementation in more detail. Then we describe the SPLASH implementation and derive the optimized SB-PRAM implementation which takes advantage of the uniform memory access time and of the multiprefix operations provided. In the last section, we present experiments and measurements for both implementations.

3.1 Task-oriented shared memory model

A *task* is a well-defined sequence of computations which are performed consecutively on *one* processor without being interrupted by computations of other tasks. A task terminates after all computations have been executed. At any point of its execution a task can create new tasks which do not need to be executed on the same processor as their creating task. A newly created task can be executed only if there are no data missing which are needed from tasks that have not terminated yet. An *executable task* is a task for which all necessary data are available and which is ready for execution. A specific initiation structure ensures that only tasks are created which are executable. Different tasks cooperate by accessing the same data structure in the shared memory.

Asynchronous accesses to the same data structure by different processors are a potential source for *non-determinism*. Simultaneous accesses to the same data structure are coordinated with `lock` and `unlock` operations that are used to protect a particular data structure. To access a data structure that can also be accessed by other tasks at the same time, a task executes a `lock` operation. Only the task initiating the lock operation is allowed to manipulate the corresponding data structure. All other tasks that try to access the data structure have to wait until the active task executes an `unlock` operation. If several tasks simultaneously execute `lock` operations on the same data structure, access is granted only to one task T and all other tasks have to wait until T executes an `unlock` operation.

A centralized *task pool* is used to store all executable tasks. New tasks are included after their creation but not before all tasks they depend on have been executed, i.e., not before all input data are available. A task is removed from the pool when it is assigned to a processor starting the execution. A common *task management* strategy is the use of task queues which can be accessed by the processors when searching for new work to do. For a small number of processors, a *central, lock-protected task queue* is appropriate if the average task length is large enough so that the probability of two processors accessing the queue at the same time is small. For a larger number of processors or smaller task lengths, *decentralized task queues* are in use where each processor has its own queue. Initially, the tasks are partitioned statically to the different queues. To avoid a bad load balance, a processor is allowed to access the task queues of the other processors, if its own queue is empty (*task stealing*) [38]. A locking mechanism must be used to manage concurrent accesses to the same queue. But this increases the task management overhead and may lead to sequentializations if the processors try to access the task queues of other processors quite often.

An alternative is the use of a central *parallel task queue* that can be accessed by several processors in parallel without sequentializations. For an efficient realization, this requires special hardware support by the machine for a coordinated access to shared data. An example for such support is the multiprefix operation provided by the SB-PRAM, as described in the following subsection.

3.2 Execution platform and software support

The SB-PRAM is an implementation of a modified fluent machine [1]. A number p of physical processors have access to p memory modules which are connected to the processors via a butterfly interconnection network. The memory is accessed as a virtual linear shared memory distributed among the modules. To avoid congestion of a memory module, logical addresses are distributed randomly by hashing to the memory modules. Each physical processor simulates a fixed number v of *virtual* processors in a pipelined way, i.e., after executing a machine instruction of a virtual processor i, a physical processor performs a context switch and executes the next machine instruction of virtual processor $(i + 1) \bmod v$. Fast context switching is supported by separate register files for each virtual processor and special control hardware. The number of virtual processors per physical processor is chosen such that, with respect to the cycle time of a virtual processor, the latency of the network is hidden with one delay cycle for a load and no delay cycles for a store operation.

The simulation of virtual processors by the physical processors is hidden from the programmer, i.e., the SB-PRAM appears as a parallel machine with $p \cdot v$ (virtual) processors and a global shared memory with uniform memory access time. This makes the SB-PRAM an ideal platform for the investigation of scalability and granularity issues of irregular applications on shared memory machines with many processors. Locality issues that are significant for other (cache-based) systems like the Stanford DASH [22] or the Convex SPP1600 [32] can be ignored. Currently, a prototype of the SB-PRAM with 128 physical processors and 128 memory modules comprising a global memory of 2GBytes is under construction. In this configuration, each physical processor simulates

32 virtual processors, thus providing a total of 4096 virtual processors. A prototype with 4 physical processors simulating 128 virtual processors is already working. In this article, investigations of the hierarchical radiosity method are carried out with a simulator that performs an exact simulation of the full machine. A comparison of the simulation results with measured runtimes of a 4-processor prototype for different irregular applications, including a particle simulation methods [27], shows that the simulation results exactly match the performance of the prototype, i.e., the 4-processor-prototype behaves like a PRAM machine and the locality or non-locality of memory access does not influence the execution time. Therefore, we expect the simulation results for more than 128 (virtual) processors to predict the behavior of the full machine accurately.

Besides the usual load and store operations to access memory cells, the SB-PRAM also offers *multiprefix* instructions that enable several processors to perform simple operations on a memory cell in parallel. We illustrate the execution of a multiprefix instruction with the multiprefix addition instruction MPADD. Let $p_1, \ldots, p_n$ be the executing processors where processor p_i contributes a local value o_i. Let s be a shared memory cell with value o. If $p_1, \ldots, p_n$ execute the MPADD operation synchronously, i.e., processor p_i executes

$$\text{MPADD } s, o_i \ ,$$

then after the operation, processor p_j holds value $o + \sum_{i=1}^{j-1} o_i$, and s contains the value $o + \sum_{i=1}^{n} o_i$. The multiprefix operations MPMAX, MPOR, and MPAND work similarly. All multiprefix operations are implemented by combining values in the network, i.e., on the path from the processors to the memory module containing s ,local values of the processors are combined according to the specified reduction operation such that the final result arrives at memory cell s. The partial values are send back appropriately to the processors. Hence, a multiprefix operation is performed in the same time as a read operation, i.e., in two time units, independently of the number of participating processors. It is even possible that different groups of processors perform separate multiprefix operations in parallel. The multiprefix operations can be used for an efficient implementation of synchronization mechanisms —like locks for concurrent memory accesses or barrier synchronizations—and for the implementation of various parallel data structures, such as priority queues or FIFO queues [18].

The multiprefix instructions can also be exploited to avoid several sources of *sequentialization* that reduces speedups. For example, in many situations the locking mechanism for a concurrent access to shared data structures can be avoided by using appropriate multiprefix operations which do not cause a sequentialization. This is especially important for a large number of processors, since the probability that two processors try to access the same data structure at the same time increases as the number of processors increases. Therefore, the sequentializations caused by locking the data structure may severely reduce the efficiency for a large number of processors.

The multiprefix operations can also be used for efficient task management without sequentialization. The overhead introduced by task stealing can be avoided by using a *central parallel task queue* which can be accessed by all processors [18]: Let t be the number of tasks to be stored in a task queue. A *parallel* task queue Q allows the simultaneous access of p (virtual) processors without sequentialization. This can be realized by using an appropriate data structure (an array with p entries each pointing to a task queue) which is accessed by a multiprefix operation. For $p \leq t$, each of the p processors gets an executable task from a separate queue. For $p > t$, only t processors get a task and $p - t$ processors are idle waiting for new tasks to be added to Q. As soon as this happens, the new tasks are assigned to the waiting processors.

3.3 SPLASH implementation

The parallel implementation realizes all phases in the specification of the algorithm in Figure 6, i.e., the construction of the BSP tree, the radiosity-based refinement, the solution of the linear system, and the final smoothing step. Parallelism occurs (i) across input polygons, (ii) across the patches that a polygon is divided into, and (iii) across the interactions computed for one patch. This parallelism is reflected in the definition of the tasks as shown Figure 7. The B-tasks and F-tasks realize parallelism in the first phase. Note that the F-tasks compute only form factors between input polygons, but form factors between patches are computed within the R-tasks. The parallel computation of interactions and parallel radiosity-based refinement is realized by R-tasks and V-tasks. The last phase is structured into A-tasks.

TASK NAME	COMPUTATIONS PERFORMED BY THE TASK
B-task(R)	insert input polygon R into BSP tree and create B- and F-tasks
F-task(i,j)	compute form factors F_{ij} and F_{ji} for input polygons R_i and R_j
R-task(q)	compute phase 2 for patch/element q except visibility factor
V-task(i,j)	compute visibility factor V_{ij} between patches/elements i and j
A-task(q)	create A-tasks for children of patch q and perform bilinear interpolation if q is an element

Fig. 7. Types of tasks in the SPLASH implementation.

In the original SPLASH-2 implementation each processor has its own task queue and inserts new tasks created by local tasks into this queue to maintain locality. But because of the dynamically changing hierarchical data structure,

```
(0) Prephase: sort input polygons R_1, ..., R_k;
(1) Phase 1:
      do recursively: B-task(R_1)
      with task B-task(R_i) = {
        Insert polygon R_i into BSP tree;
        do in parallel { B-task(R_{i+1}) || forall R_j with j<i do F-task(i,j) }
      }
(2) Phase 2:
      do {
        forall polygons R ∈ {R_1, ... R_k} do recursively: R-task(root(R));
        with task R-task(j) = {
          forall interactions i ∈ I(j) do {
            V-task(j,i);
            if (V_ji F_ji B_i > B_ε and A_j, A_i > A_ε) then {
              Divide (A_j, A_i);
              Compute form factor for each new interaction;
            }
          }
          Compute B_j = ρ_j Σ_{i∈I(j)} V_ji F_ji B_i + B_parent(j);
          if j is leaf then {
            B_j = B_j + E_j;
            while (radiosity values of all children of j
                              have been computed) do {
              j = parent(j);
              B_j = 1/4 Σ_{i=child(j)} B_i;
            }
          }
          else { forall children r of j do R-task(r); }
        }
      } while (ERROR > ε )
(3) Phase 3:
      forall polygons R ∈ {R_1, ... R_k} do recursively: A-task(root(R));
      with task A-task(q) = {
        forall children r of q do A-task(r);
        if (q is leaf) then perform bilinear interpolation for edges; }
```

Fig. 8. Task organization of the SPLASH implementation.

load balance cannot be achieved by a static assignment of tasks, so task stealing is used to improve the load balance. Simultaneous accesses to the same data are avoided by the locking mechanism, e.g., when interactions between patches/elements are computed.

The task program of the SPLASH implementation, illustrated in a pseudo-code task-program in Figure 8, represents one possible implementation of the program in Figure 6. We have chosen a task description in a pseudo-code language reflecting the fact that tasks perform computations and also initiate other tasks (similarly, procedures in sequential programs perform computations and call other procedures). (The pseudo-code language is similar to the

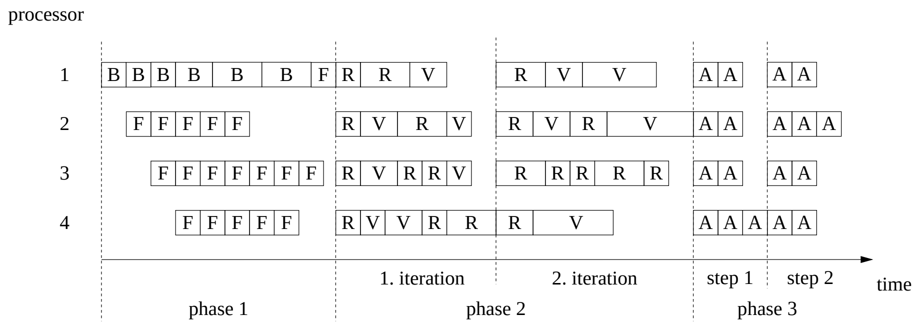

Fig. 9. Task scheduling for SPLASH tasks.

pseudocode language used in Figure 6.) The B-tasks for inserting polygons into the BSP tree (see Figure 7) have to be executed sequentially. The initiation of data dependent F-tasks is expressed by a recursive call-structure using the keyword do recursively. The corresponding task is defined by a with task statement having a recursive structure due to the hierarchical tree structure used in the algorithm. A possible schedule of B-tasks and F-tasks on 4 processors is depicted in Figure 9. The computations on different quadtrees in phase (2) are independent, as expressed by a forall construct. The interactions within each quadtree are performed recursively according to the tree structure determining data dependencies between R-tasks. The visibility V-tasks for children are initiated by the parents' R-tasks. The last phase creates a hierarchy of tasks of which only the leaf-tasks perform the bilinear interpolation.

3.4 SB-PRAM implementation

Optimizations of the parallel implementation include the parallel construction of the BSP tree, the use of a parallel task queue, and the use of parallel loops where locality can be ignored. The changes concern all three phases of the algorithm. Figure 10 summarizes the modifications.

The BSP tree is constructed by a parallel search over the polygon tree in contrast to a sequential construction in order to reduce the sequential portions of the computation. Because of Amdahl's law, this is especially important for large numbers of processors. In the second phase, the tasks to compute interactions (one task for each input polygon) do not offer enough parallelism for a large number of processors. In the first iteration step the number of tasks cannot be increased. But because these interactions all take place on the same level, this phase is separated from the rest of the iteration and the mutual configuration factors are computed with a parallel loop. Moreover, the symmetry of the configuration factors is exploited. In all following iterations, the computations of configuration factors and visibility factors are moved to lower levels in the quadtrees, creating a high degree of parallelism. The tasks for the final bilinear interpolation to smooth the solution are also executed in

PHASE	MODIFICATION	TASKS	ADVANTAGES
1	algorithmic modification	redefined B'-tasks for building BSP tree	larger potential parallelism; worthwhile for a large number of processors;
1, 2	algorithmic modification and change of tasks	parallel loop for combined initial interactions V_{ij}, F_{ij}	creation of regular `forall`-loop realized by *parallel loop*; exploitation of form-factor symmetries is possible
2	change of tasks	modified V'-tasks for computing V_{ij}, F_{ij}	refinement of granularity
3	implementation	*parallel loop* for bilinear interpolation	task-administration for `forall`-loop is avoided
1-3	software support	task allocation	parallel queue avoids failures in task stealing

Fig. 10. Optimizations for an efficient SB-PRAM implementation.

a parallel loop over the leaf elements. This strategy, which replaces the version where all internal patch nodes were involved in creating tasks for their child nodes, improves locality. The modified tasks are described in Figure 11.

TASK NAME	COMPUTATIONS PERFORMED BY THE TASK
B'-task(R)	insert polygon R into BSP tree and build sublists of elements
F'-comp.(i,j)	compute form factors F_{ij}, F_{ji} and visibility factors V_{ij}, V_{ji} for input polygons R_i and R_j
R'-task(q)	compute phase 2 for element/patch except form and visibility factors
V'-task(i,j)	compute visibility factor V_{ij} and form factor F_{ij} between patches/elements i and j
A'-comp.(q)	compute bilinear interpolation for element q

Fig. 11. Types of tasks in the SB-PRAM implementation.

The modified task program of the hierarchical radiosity algorithm with radiosity based subdivision is given in Figure 12, showing four phases (instead of three) which are still separated by synchronization points. The first phase (1') constructs the BSP tree in parallel with the help of modified B'-tasks that compute independent parts of the BSP tree in parallel, yielding the same BSP tree as with the sequential execution. The computation of form factors and visibility factors for input polygons is executed in the second phase (1"), i.e., phase (1") executes parts of the phases (1) and (2) from the SPLASH implementation. The recursive subdivision and the iterative computation of radiosity values is executed in phase (2') which uses V'-tasks to compute form factors and visibility factors. Phase (3') executes the same computations as phase (3), but the bilinear interpolation is directly applied to the elements of the quadtrees. The implementation supports the use of parallel loops and

```
(0)  Prephase: sort input polygons R_1, ... R_k;
(1')  Phase 1':
        do recursively: B'-task(R_1, ..., R_k);
        with task B'-task(R_1, ..., R_k) = {
           Insert R_1 in BSP tree;
           Build lists of polygons (q_1, ..., q_l) visible
           and (r_1, ..., r_m) invisible from R_1;
           do in parallel {B'-task(q_1, ..., q_l) || B'-task(r_1, ..., r_m) } }
(1")  Phase 1":
        forall i, j = 1, ..., k with i < j do F'-computation(i,j);
(2')  Phase 2':
        do {
           forall polygons R ∈ {R_1, ..., R_k} do recursively: R'-task(root(R));
           with task R'-task(j) = {
              forall interactions i ∈ I(j) do {
                 V'-task(j,i);
                 if (V_ji F_ji B_i > B_ε and A_j, A_i > A_ε) then {
                    Divide (A_j, A_i);
                 }
              }
              Compute B(j) = ρ_j Σ_{i∈I(j)} V_ji F_ji B(i) + B(parent(j));
              if j is leaf then {
                 B(j) = B(j) + E(j);
                 while (radiosity values of all children of j
                                  have been computed) do {
                    j = parent(j);
                    B(j) = 1/4 Σ_{i=child(j)} B(i);
                 }
              }
              else { forall children r of j do R'-task(r);}
           }
        } while (ERROR > ε)
(3')  Phase 3':
        forall elements q in all quadtrees do A'-computations(q);
```

Fig. 12. Task organization for the SB-PRAM.

increases the task granularity. A possible schedule on 4 processors is given in Figure 13.

4 Experiments

The SPLASH2 benchmark suite comprises several parallel example programs realizing irregular applications, which are mainly intended for the Stanford DASH multiprocessor [23,30,31]. The DASH is a shared address space machine with cache coherency, physically distributed memory, and non-uniform

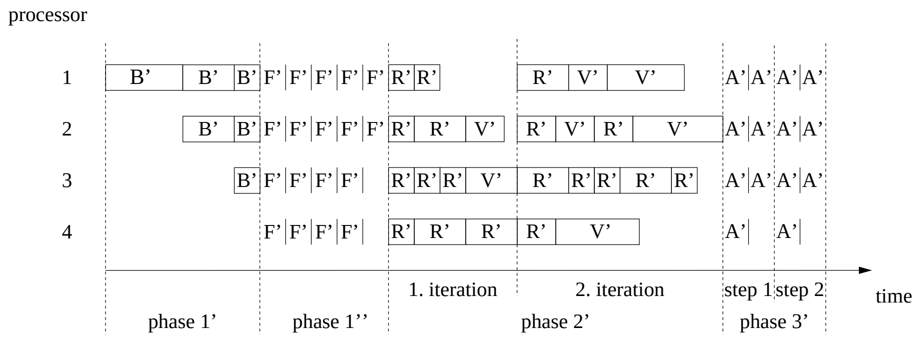

Fig. 13. Task scheduling for SB-PRAM-tasks.

memory access time. The prototype has 64 processors that are organized in 16 clusters. A cluster comprises four MIPS R3000 processors and a memory module, all are connected by a shared bus. The clusters are connected in a mesh network. The memory hierarchy consists of registers, first level cache, second level cache, main memory local to cluster, and remote main memory in another cluster. The access to the first level cache is two orders of magnitude faster than the access to remote main memory. Therefore, data locality plays an important role in attaining a good performance.

Figure 14 (left) shows the speedups of the original SPLASH implementation on the DASH and on the SB-PRAM simulator, using the SPLASH test scene. For the implementation on the DASH, the default task granularity was used. For the implementation on the SB-PRAM, the figure shows the speedups for the default task granularity and the finest garnularity that is possible. The SPLASH-2 scene contains 346 input polygons which are decomposed into 12430 elements and for which 150572 interactions between patches have to be computed. The number of B-tasks used during the computation is 372, i.e., eight input polygons had to be split. The number of F-tasks is 17202, the number of R-tasks is 126332, the number of V-tasks is 57324, and the number of A-tasks is 32944. The speedup values for the DASH for up to 48 processors are reported in [30]. Due to the task stealing mechanism, the DASH is most efficient with coarser granularity where the V-tasks are chosen to compute four visibility values instead of one; this improves locality on the BSP tree data structure (DASH(default) in Figure 14 (left)). In contrast, the SB-PRAM achieves a better speedup when a finer granularity with one visibility computation per V-task is used (SB-PRAM(finest) in Figure 14 (left)). The original SPLASH implementation performs better on the SB-PRAM (SB-PRAM(default) in Figure 14 (left)) than on the DASH because of uniform memory access times (see Figure 9 on the left) and the redundance of locality.

The large number of processors and the additional software support (see Section 3) make the SB-PRAM an ideal platform on which for studying the scalability properties inherent in an algorithms. But the massive parallelism and the uniform access time require a different implementation strategy for the study

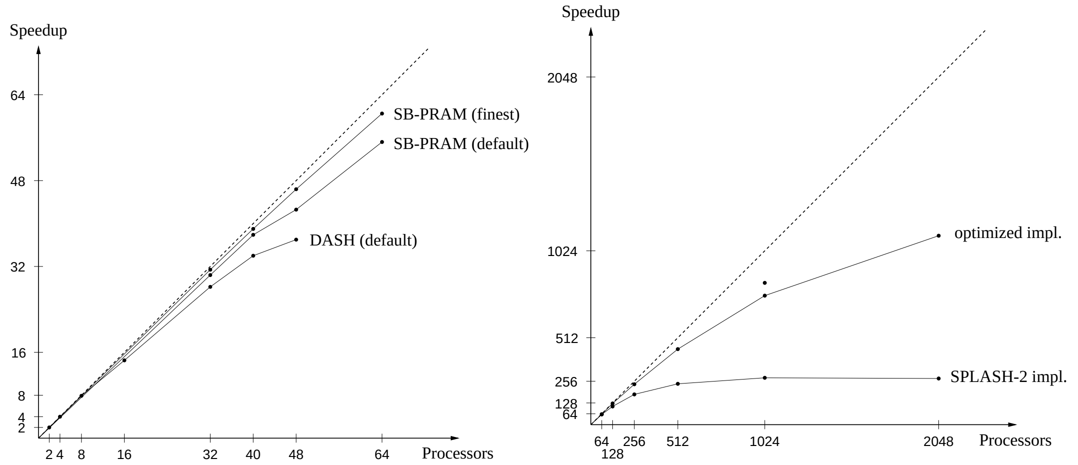

Fig. 14. Speedups for the radiosity application on the SB-PRAM and on the DASH using the SPLASH-2 implementations.

of the scalability of the hierarchical radiosity method. The optimizations summarized in Figure 10 take this modified approach into account by replacing the expensive task concept by *parallel loops* if the loops exhibit a regular, independent parallel structure (F'-computations, A'-computations), decreasing the granularity (new V'-tasks and R'-tasks with smaller runtime), increasing the degree of potential parallelism by destroying data locality (B'tasks), and exploiting a new task-administration mechanism by using unit access time task-queues.

Figure 14 (right) shows the speedups for the original SPLASH implementation and the optimized implementation on the SB-PRAM for up to 2048 processors. The SPLASH implementation with the default granularity does not scale well for more than 256 processors. For 1024 processors the optimized version with the finest task granularity still has an efficiency of 0.742. Figure 15 reports the speedup value and the absolute runtimes in seconds for both implementations applied to the SPLASH-2 test scene. The speedup values for the SPLASH implementation with phases (1), (2), (3) are on the left; the values for the optimized version with phases (1'), (1"), (2'), (3') are on the right. The phases overlap according to the algorithmic structure. The timings in columns T_1 left and right show the optimized version has a sequential runtime which is 5.8% faster than the non-optimized version. The efficient parallelization of the B-tasks (BSP tree) is more important for massively parallel implementations than for a relatively small number of processors, where the fraction $\frac{1}{14000}$ of the computation for building the BSP tree is negligible. For the SPLASH test scene with 346 input polygons, the parallel construction of the BSP tree has only a small advantage over a sequential construction. This is shown by the small speedup value of 1.44 in Figure 15. For larger test scenes, the speedup of this phase may increase significantly, e.g., for a test scene with 7048 input polygons, a parallel construction of the BSP tree achieves a speedup of 6.8 which is already obtained for 24 processors. The prephase for sorting the input polygons is done before building the BSP tree. Experiments have shown that the global execution time of the entire algorithm depends significantly on the order of the input polygons; the time varies by a factor of up to 10. This is a general phenomenon which should be separated from issues of parallelization.

A different phenomenon concerns the convergence of the iteration steps solving the system of radiosity values (4). The iteration may converge faster in the parallel implementation than in the sequential one, saving a full iteration step. The reason is the use of updated values within one iteration step if the number of patches and elements is higher than the number of processors. The faster convergence corresponds to the fact that the Gauss-Seidel method for solving linear equation systems converges faster than the Jacobi method. This effect is exploited in the (non-hierarchical) progressive radiosity method [8].

Besides the improvements of the efficiency, our investigations show that the optimized version leads to a much simpler source code. The main reasons are the lack of locality, the use of *parallel loops* which corresponds to the loops in the pseudocode algorithms, and the simplified task administration.

Figures 16 and 17 show different stages in the visualization of a scene with 1059 input triangles that are decomposed into 80911 elements. For this test scene, 224774 interactions have to be computed. The SB-PRAM implementation reaches an efficiency of 0.816 on 1024 processors.

<table>
<tr><th colspan="5">SPLASH2 implementation</th><th colspan="5">optimized implementation</th></tr>
<tr><td></td><td>Phase</td><td>T_1</td><td>T_{1024}</td><td>S_{1024}</td><td>S_{1024}</td><td>T_{1024}</td><td>T_1</td><td>Phase</td><td></td></tr>
<tr><td>dimension</td><td></td><td>sec</td><td>sec</td><td></td><td></td><td>sec</td><td>sec</td><td></td><td></td></tr>
<tr><td rowspan="2">BSP and form factor</td><td rowspan="2">1</td><td rowspan="2">6898.7</td><td rowspan="2">104.6</td><td rowspan="2">66.0</td><td>1.44</td><td>5.409</td><td>7.805</td><td>1'</td><td>BSP</td></tr>
<tr><td>876.3</td><td>7.413</td><td>6496.2</td><td>1"</td><td>root inter.</td></tr>
<tr><td>iteration</td><td>2</td><td>90726</td><td>244.1</td><td>371.7</td><td>791.5</td><td>108.0</td><td>85487</td><td>2'</td><td>iteration.</td></tr>
<tr><td>bil.interp.</td><td>3</td><td>33.782</td><td>4.225</td><td>8.0</td><td>814.9</td><td>0.036</td><td>29.336</td><td>3'</td><td>bil.interp.</td></tr>
<tr><td>total</td><td></td><td>97658</td><td>353.2</td><td>276.5</td><td>760.5</td><td>121.0</td><td>92021</td><td></td><td>total</td></tr>
</table>

Fig. 15. Timings of the SPLASH2 implementation and the optimized implementation on the SB-PRAM. The input scene is the SPLASH2 test scene.

5 Related Work

Several approaches for parallel execution of the radiosity method have been considered in the past. Most of this work has been applied to non-hierarchical variants, in particular the progressive radiosity algorithm. A good overview can be found in [39], see also [12]. A parallel gathering method for a transputer-based network using a master-slave model is given in [25]. For a small test scene, the method shows an efficiency of 50% on 24 slave processors. Parallel implementations of the progressive radiosity method have been described for different target architectures like distributed memory MIMD [4,6,10,19,33,34] shared memory MIMD [3], SIMD [9,36], and networks of workstations [7], and execution models like master-slave [4,7,10], producer-consumer [3], and SPMD [19,34], see [34,39] for a detailed overview.

Prior work on parallel implementations of the hierarchical radiosity method is described in [5,12,14,13,30,39]. The implementation from [30] was the starting point for the implementation in this article and has been discussed in the previous sections. Zareski [39] describes several implementations of the hierarchical radiosity method on networks of workstations (SP-1), virtual shared memory machines (KSR-1) and distributed memory machines (CM-5) with active messages. In contrast to the approach in this article, those implementations use a master-slave model and are directed towards exploiting coarse-grained parallelism. A single master process controls the overall calculation and farms out work for the slaves which are mainly used to perform visibility determinations between patches of the scene. The master collects the results, computes the form factors and updates the radiosities. An implementation on different architectural platforms shows that the communication overhead is too large to yield good performance. The execution time grows with increasing numbers of processors.

Funkhouser [12] presents a group iterative method based on a master-slave model which also aims at the exploitation of coarse-grained parallelism. The master repeatedly partitions patches into groups, distributes a copy of each group to a slave processor for an update of the radiosities for all patches in that group, and merges the updated radiosities back into the master solution. The groups are chosen such that the number g of groups is considerably larger than the number p of slave processors ($g > 8p$). On the other hand, each group must be large enough that the time required to distribute its computation to a slave is much smaller than it would have been to perform the computation locally on the master. The parallel implementation has been applied to a test scene with 6418 patches, 242 of which belong to light sources. The execution on a workstation cluster with up to eight Silicon Graphics slave workstations shows an efficiency between 65 and 75%. The efficiency is limited due to the synchronous master-slave communication, i.e., the master can only talk to one slave at a time. This leads to sequentialization if two slaves finish their computation at the same time and try to report the result to the master. Thus, the approach is not scalable to a large number of processors.

Garmann and Müller propose in [14,13] an SPMD approach suitable for an execution on distributed memory machines. The basic idea is to understand the radiosity computations as a manipulation of a task graph. The nodes represent tasks to compute interactions between patches and to subdivide patches into subpatches. The edges represent data exchanges between tasks. The nodes are weighted with an expected computation time, the edges with an expected communication time. The goal is to distribute the nodes among the processors so that the communication between the processors is minimized and each processor has to perform an equal amount of work. An approximation to the optimal solution is obtained with simulated annealing. An implementation on a CM-5 with 64 processors leads to a speedup of 8.4 for a test scene with 324 polygons and 6380 elements. The speedup is limited mainly by the amount of communication.

6 Conclusions

We have presented a task oriented shared memory implementation for the hierarchical radiosity method, a visualization method from computer graphics. The hierarchical radiosity method is realized by an irregular algorithm concerning dynamically changing data structures, data accesses, and different demands for computations. The main interest was to investigate the scalability issues of the method. The SB–PRAM with uniform access time offers a good platform to study efficient implementations and scalability properties for irregular problems because the locality properties of the applications do not influence the resulting performance and the investigations can concentrate on the maximum degree of parallelism. The experiments have shown that an implementation designed for up to 64 processors is not suitable for achieving good speedups for large numbers of processors.

To overcome the limitations, we introduced optimizations to exploit the facilities of the execution platform. This includes a redesign of the algorithms so as to provide a large number of independent tasks. The means are regular forall-loops and a decrease of the granularity which could be achieved in the phases of the hierarchical radiosity algorithm realizing the interactions between different surfaces. The parallel algorithms presented shows good speedup for up to 2048 processors. Thus, the hierarchical radiosity methods can be implemented efficiently, although it has highly irregular computation and access patterns. Moreover, the investigations have shown that parallel data structures provided by the underlying machine support the massively parallel, efficient implementations of highly irregular algorithms for which the hierarchical radiosity is a good example.

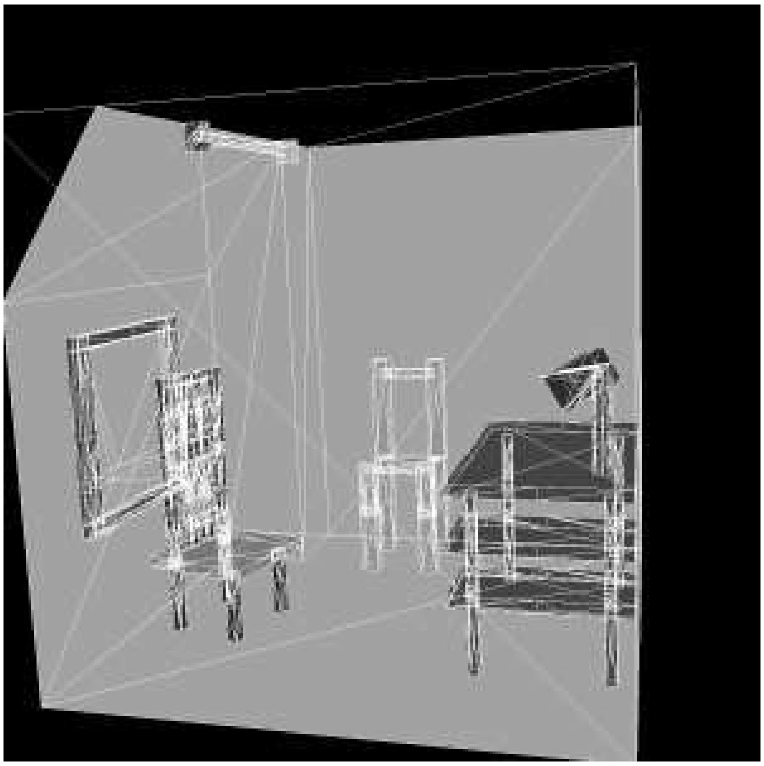

Fig. 16. Environment with 1059 input triangles.

7 Acknowledgement

We would like to thank John O'Donnell for many interesting and useful discussions. We are also grateful to the SB-PRAM software team, Saarbrücken, Germany, for the development of the simulator, the operating system, the compiler, and for the implementation of basic parallel data structures.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost–Effectiveness of PRAMs. In *Proceeding of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1991.

[2] A. Appel. An efficient program for many–body simulation. *SIAM J. Sci. Stat. Computing*, 6(1):85–103, 1985.

[3] D.R. Baum and J.M. Winget. Real-Time Radiosity through Parallel Processing and Hardware Acceleration. *Computer Graphics*, 24(1):67–75, 1990. Proceedings of SIGGRAPH '90.

[4] K. Bouatouch and T. Priol. Data Management Scheme for Parallel Radiosity. *Computer-Aided Design*, 26(12):876–882, 1994.

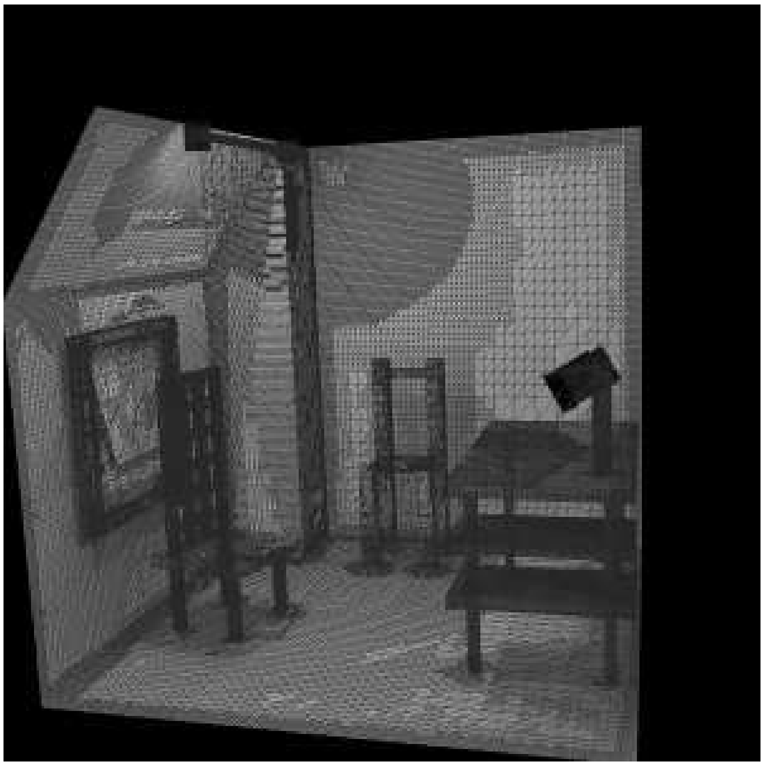

Fig. 17. Hierarchical subdivision of the environment in Figure 16 into 80991 elements.

[5] M.B. Carter. *Parallel Hierarchical Radiosity Rendering.* PhD thesis, Iowa State University, 1993.

[6] A. Chalmers and D. Paddon. Parallel Processing of Progressive Refinement Radiosity Methods. In *Proceedings of the 2nd Eurographics Workshop on Rendering*, 1991.

[7] S.E. Chen. A Progressive Radiosity Method and its Implementation in a Distributed Procesing Environment. Master's thesis, Cornell University, 1989.

[8] M. Cohen, Sh. Chen, J. Wallace, and D. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, 1988. Proceedings of the SIGGRAPH '88.

[9] S.M. Drucker and P. Schroeder. Fast Radiosity Using a Data Parallel Architecture. In *Proceedings of the 3nd Eurographics Workshop on Rendering*, pages 247–258, 1992.

[10] M. Feda and W. Purgathofer. Progressive Refinement Radiosity on a Transputer Network. In *Proceedings of the 2nd Eurographics Workshop on Rendering*, pages 139–148, 1991.

[11] J. Foley, A. van Dam, St. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice.* Addison–Wesley, Reading, USA, 1990.

[12] T.A. Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. *Computer Graphics*, 30(4), 1996. Proceedings of SIGGRAPH '96.

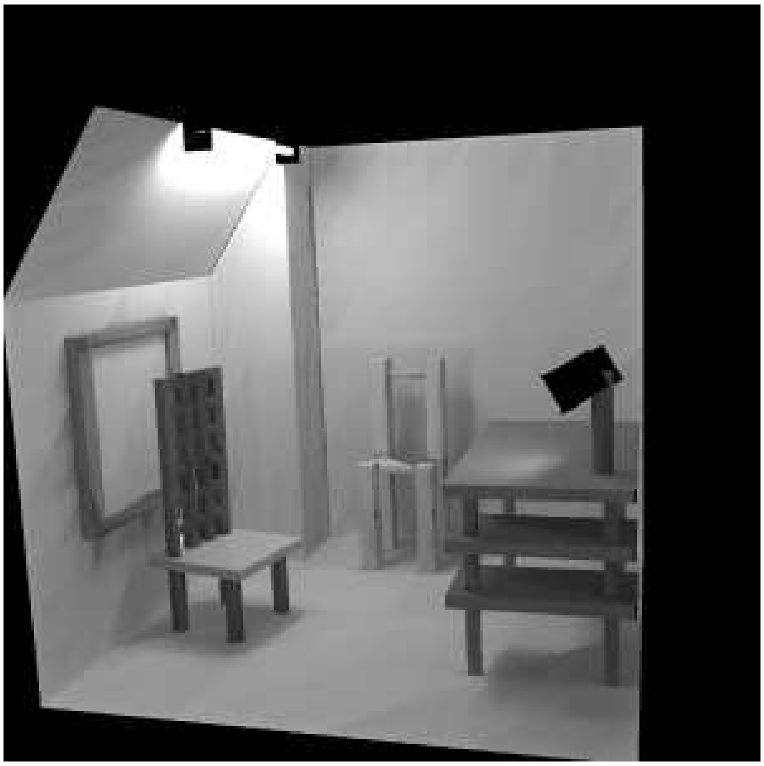

Fig. 18. Visualization of the computed radiosities of the elements.

[13] R. Garmann and C.A. Bohn. A Parallel Approach to Hierarchical Radiosity. In *Proceedings of the Winter School of Computer Graphics and CAD Systems*, pages 26–35, 1995.

[14] R. Garmann, C.A. Bohn, and H. Müller. Parallel Hierarchical Radiosity on the CM-5. Technical Report 557/1994, University of Dortmund, 1994.

[15] A. S. Glassner. *An Introduction to Ray Tracing.* Academic Press, 1989.

[16] C.M. Goral, K.E. Torrance, D.P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):212–222, 1984. Proceedings of the SIGGRAPH '84.

[17] L. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* MIT Press, Boston, 1988.

[18] Th. Grün, Th. Rauber, and J. Röhrig. The programming environment of the SB-PRAM. In *Proc. 7th IASTED/ISMM Int.l Conf. on Parallel and Distributed Computing and Systems, Washington DC*, October 1995.

[19] P. Guitton, J. Roman, and G. Subrenat. Implementation Results and Analysis of a Parallel Progressive Radiosity. In *1995 Parallel Rendering Symposium*, pages 31–37, 1995.

[20] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, 1991. Proceedings of SIGGRAPH '91.

[21] P.S. Heckbert. *Simulating Global Illumination using Adaptive Meshing.* PhD thesis, university of California, Berkeley, 1991.

Fig. 19. Gouraud shaded visualization with texture mapping.

[22] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Trans. on Parallel and Distributed Systems*, 4 (1):41–61, 1993.

[23] D. Lenoski, J. Laudon, T. Joe D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, 1993.

[24] D.E. Lenoski and W. Weber. *Scalable Shared-Memory Multiprocessors.* Morgan Kaufmann, 1995.

[25] W. Purgathofer and M. Zeiller. Fast radiosity by Parallelization. In *Proceedings of the Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 173–183, 1990.

[26] Th. Rauber. *Algorithmen in der Computergraphik.* Teubner Verlag, 1993.

[27] Th. Rauber, G. Rünger, and C. Scholtes. Shared-memory Implementation of an Irregular Particle Simulation Method. In *Proceedings of the EuroPar'96*, Springer LNCS 1123, pages 822–827, 1996.

[28] R. Siegel and J. Howell. *Thermal Radiation Heat Transfer.* Hemisphere Publishing Corporation, Washington, USA, 1981.

[29] J.P. Singh. *Parallel Hierarchical N-Body Methods and their Implication for Multiprocessors.* PhD thesis, Stanford University, 1993.

[30] J.P. Singh, A. Gupta, and M. Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(4):45–55, 1994.

[31] 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*, 27:118–141, 1995.

[32] T. Sterling, D. Savarese, P. Merkey, and K. Olson. An Enpirical Study of the Convex SPP-1000 Hierarchical Shared-Memory System. In *Proceedings of the PACT'95*, 1995.

[33] D. Stuttard, A. Worrall, D. Paddon, and C. Willis. A Parallel Radiosity System for Large Data Sets. In *Proceedings of the Winter School of Computer Graphics and CAD Systems*, pages 421–430, 1995.

[34] G. Subrenat. *Parallélisation de Méthodes de Radiosité à Base de Lancer de Rayons*. PhD thesis, University of Bordeaux, 1995.

[35] W. Thibault and B. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics*, 21(4):153–162, 1987. Proceedings of the SIGGRAPH '87.

[36] A. Varshney and J.F. Prins. An Environment Projection Approach to Radiosity for Mesh-connected Computers. In *Proceedings of the 3nd Eurographics Workshop on Rendering*, pages 271–281, 1992.

[37] J. Wallace, K. Elmquist, and E. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):315–324, 1989. Proceedings of the SIGGRAPH '89.

[38] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[39] D. Zareski. Parallel Decomposition of View-independent Global Illumination Algorithms. Master's thesis, Cornell University, 1996.