The Visual Computer

A parallel scaled conjugate-gradient algorithm for the solution phase of gathering radiosity on hypercubes

Tahsin M. Kurç, Cevdet Aykanat, Bülent Özgüç

Department of Computer Engineering and Information Science, Bilkent University, 06533, Bilkent, Ankara, Turkey

Gathering radiosity is a popular method for investigating lighting effects in a closed environment. In lighting simulations, with fixed locations of objects and light sources, the intensity and color and/or reflectivity vary. After the form-factor values are computed, the linear system of equations is solved repeatedly to visualize these changes. The scaled conjugate-gradient method is a powerful technique for solving large sparse linear systems of equations with symmetric positive definite matrices. We investigate this method for the solution phase. The nonsymmetric form-factor matrix is transformed into a symmetric matrix. We propose an efficient data redistribution scheme to achieve almost perfect load balance. We also present several parallel algorithms for form-factor computation.

Key words: Gathering radiosity – Scaled conjugate-gradient method – Parallel algorithms – Hypercube multicomputer – Data redistribution

Correspondence to: T.M. Kurç

1 Introduction

Realistic synthetic image generation by computers has been a challenge for many years in the field of computer graphics. It requires the accurate calculation and simulation of light propagation and global illumination effects in an environment. The radiosity method (Goral et al. 1984) is one of the techniques for simulating light propagation in a closed environment. Radiosity accounts for the diffuse inter-reflections between the surfaces in a diffuse environment. There are two approaches to radiosity: *progressive refinement* (Cohen et al. 1988) and *gathering*. Gathering is a very suitable approach for investigating lighting effects within a closed environment. For such applications, the locations of the objects and light sources in the scene usually remain fixed, while the intensity and color of light sources and/or reflectivity of surfaces change in time. The linear system of equations is solved many times to investigate the effects of these changes. Therefore, efficient implementation of the solution phase is important for such applications.

Although gathering is excellent for some applications in realistic image generation, it requires much computing power and memory storage to hold the scene data and computation results. As a result, applications of the method on conventional uniprocessor computers for complex environments can be far from practical due to high computation and memory costs. Distributed memory multicomputers, however, can provide a cost-effective solution to problems that require much computation power and memory storage. In these types of architectures, processors are connected to other processors by an interconnection network, such as a hypercube, ring, mesh, etc. Data are exchanged and processors are synchronized via message passing.

Various parallel approaches have been proposed and implemented for gathering radiosity (Chalmers and Paddon 1989, 1990; Price and Truman 1990; Paddon et al. 1993). In these approaches, the Gauss–Jacobi (GJ) method is used in the solution phase. The scaled conjugate-gradient (SCG) method is known to be a powerful technique for the solution of large sparse linear systems of equations with symmetric positive definite matrices. In general, the SCG method converges much faster than the GJ method. In this work, the utilization and parallelization of the SCG method is investigated for the solution phase. The nonsymmetric

The Visual Computer (1997) 13:1–19

form-factor matrix is efficiently transformed into a symmetric matrix. An efficient data redistribution scheme is proposed and discussed to achieve an almost perfect load balance in the solution phase. Several parallel algorithms for the form-factor computation phase are also presented.

The organization of the paper is as follows. Section 2 describes the computational requirements and the methods used in the form-factor computation and solution phases. The GJ and SCG methods for the solution phase are described in this section. Section 3 briefly summarizes the existing work on the parallelization of the radiosity method. Section 4 briefly describes the hypercube multicomputer. Section 5 presents the parallel algorithms for form-factor computation. The parallel algorithms developed for the solution phase are presented and discussed in Sect. 6. Load balancing in the solution phase and a data redistribution scheme are discussed in Sect. 7. Finally, experimental results from a 16-node Intel iPSC/2 hypercube multicomputer are presented and discussed in Sect. 8.

2 Gathering radiosity

In the radiosity method, every surface and object constituting the environment is discretized into small patches, which are assumed to be perfect diffusers. The algorithm calculates the radiosity value of each patch in the scene.

The gathering radiosity method (the term *radiosity method* will also be used interchangeably to refer to the gathering method) consists of three successive computational phases: the form-factor computation phase, the solution phase, and the rendering phase. The form-factor matrix is computed and stored in the first phase. In the second phase, a linear system of equations is formed and solved for each color band (e.g., red, green, blue) to find the radiosity values of all patches for these colors. In the last phase, results are rendered and displayed on the screen. They are derived from the radiosity values of the patches computed in the second phase. Conventional rendering methods (Watt 1989; Whitman 1992) (e.g., Gouraud shading, Z-buffer algorithm) are used in the last phase to display the results.

This section describes the computational requirements and the methods used in the form-factor computation and solution phases.

2.1 Form-factor computation phase

In an environment discretized into N patches, the radiosity b_i of each patch 'i' is computed as follows:

$$b_i = e_i + r_i \sum_{j=1}^{N} b_j F_{ij} \tag{1}$$

where e_i and r_i denote the initial radiosity and reflectivity values, respectively, of patch 'i', and the form-factor F_{ij} denotes the fraction of light that leaves patch 'j' and is incident on patch 'i'. The F_{ij} values depend on the geometry of the scene, and they remain fixed as long as the geometry of the scene remains unchanged. The F_{ii} values are taken to be zero for convex patches. An approximation method to calculate the form factors, called the hemicube method, is proposed by Cohen and Greenberg (1985). In this method, a discrete hemicube is placed around the center of each patch. Each face of the hemicube is divided into small squares (surface squares). A typical hemicube is composed of $100 \times 100 \times 50$ such squares. Each square 's' corresponds to a delta form-factor $\Delta f(s)$.

After allocating a hemicube over a patch 'i', all other patches in the environment are projected onto the hemicube for hidden patch removal. Then, each square 's' allocated by patch 'j' contributes $\Delta f(s)$ to the form-factor F_{ij} between patches 'i' and 'j'. At the end of this process, the ith row of the form-factor matrix $\mathbf{F}$ is constructed. The $\mathbf{F}$ matrix is a sparse matrix because a patch may not see all the patches in the environment due to the occlusions. In order to reduce the memory requirements, space is allocated dynamically for only nonzero elements of the matrix during the form-factor computation phase. Each element of a row of the matrix is in the form [*column id*, *value*]. (This compressed form requires 8 bytes for each nonzero entry, 4 bytes for the *column-id*, and 4 bytes for the *value*. We observed that $\approx 30\%$ of the matrix entries are nonzero in our test scenes. Hence, for N patches,

approximately $2.4N^2$ bytes are required to store the **F** matrix.) The *column id* indicates the j index of an F_{ij} *value* in the ith row.

2.2 Solution phase

In this phase, the linear system of equations of the form

$$\mathbf{Cb} = (\mathbf{I} - \mathbf{RF})\mathbf{b} = \mathbf{e} \tag{2}$$

is solved for each color band. Here, **R** is the diagonal reflectivity matrix, **b** is the radiosity vector to be calculated, **e** is the vector representing the self-emission (initial emission) values of patches, and **F** is the form-factor matrix.
Methods for solving such a linear system of equations can be grouped as direct methods and iterative methods (Golub and van Loan 1989). In this work, iterative methods have been used in the solution phase because they exploit and preserve the sparsity of the coefficient matrix. In addition, unlike direct methods, maintaining only **F** is sufficient in the formulation of iterative methods in this work. Hence, they require less storage than direct methods. Furthermore, iterative methods are more suitable for parallelization. It has been experimentally observed that iterative methods converge quickly to acceptable accuracy values.
Three popular iterative methods widely used for solving linear system of equations are the Gauss–Jacobi (GJ), Gauss–Seidel (GS), and conjugate-gradient (CG) methods (Golub and van Loan 1989). The GS scheme is inherently sequential; hence, it is not suitable for parallelization. Thus, only the GJ and CG schemes are described and investigated for parallelization in this work.

2.2.1 The GJ method

In the GJ method, the iteration equation for the solution phase of the radiosity becomes

$$\mathbf{b}^{k+1} = \mathbf{RFb}^k + \mathbf{e}. \tag{3}$$

Note that it suffices to store only the diagonals of the diagonal **R** matrix. Hence, matrix and vector will be used interchangeably to refer to a diagonal matrix. The GJ algorithm necessitates storing only the original **F** matrix and the reflectivity vector for each color in the solution phase. The algorithm for the GJ method is given in Fig. 1. The computational complexity of an individual GJ iteration is:

Initially, choose $\mathbf{b}^0$
for k = 1,2,3,...

1. form $\mathbf{b}^{k+1} = \mathbf{RFb}^k + \mathbf{e}$ as
   $\mathbf{x} = \mathbf{Fb}^k$; $\mathbf{y} = \mathbf{Rx}$; $\mathbf{b}^{k+1} = \mathbf{y} + \mathbf{e}$
2. $\mathbf{r}^k = \mathbf{b}^{k+1} - \mathbf{b}^k$
3. check $\mathrm{Norm}(\mathbf{r}^k)/max(\mathbf{b}^k) < \epsilon$
   where $\mathrm{Norm}(\mathbf{r}^k) = \sum_{i=1}^{N} |r_i^k|$ and $max(\mathbf{b}^k) = max(|b_i^k|)$

Fig. 1. Basic steps of the GJ method

$$T_{GJ} \approx (2M + 6N)t_{calc} \tag{4}$$

where M is the total number of nonzero entries in the **F** matrix, and N is the number of patches in the scene. Here, scalar addition, multiplication, and absolute value operations are assumed to take the same amount of time t_{calc}.

2.2.2 The SCG method

The convergence of the CG method (Hestenes and Stiefel 1952) is guaranteed only if the coefficient matrix **C** is symmetric and positive definite. However, the original coefficient matrix is not symmetric since $c_{ij} = r_i F_{ij} \neq r_j F_{ji} = c_{ji}$. Therefore, the CG method cannot be used in the solution phase using the original **C** matrix as is also mentioned by Paddon et al. (1993). However, the reciprocity relation $A_i F_{ij} = A_j F_{ji}$ between the form-factor values of the patches can be exploited to transform the original linear system of equations in Eq. 2 into

$$\mathbf{Sb} = \mathbf{De} \tag{5}$$

with a symmetric coefficient matrix $\mathbf{S} = \mathbf{DC}$ where **D** is a diagonal matrix $\mathbf{D} = diag[A_1/r_1, A_2/r_2, \ldots, A_N/r_N]$. Note that matrix **S** is symmetric since $s_{ij} = A_i F_{ij} = A_j F_{ji} = s_{ji}$ for $j \neq i$. The ith row of the matrix **S** has the following structure:

$$S_{i*} = [-A_i F_{i1}, \ldots, -A_i F_{i,i-1}, A_i/r_i, -A_i F_{i,i+1}, \ldots, -A_i F_{iN}]$$

for $i = 1, 2, \ldots, N$. Therefore, matrix $\mathbf{S}$ preserves diagonal dominance. Thus, the coefficient matrix $\mathbf{S}$ in the transformed system of equations (Eq. 5) is positive definite since diagonal dominance of a matrix ensures its positive definiteness, which is also shown by Neumann (1994, 1995) independently of our work.

The convergence rate of the CG method can be improved by preconditioning. In this work, simple yet effective *diagonal scaling* is used for preconditioning the coefficient matrix $\mathbf{S}$. In this preconditioning scheme, rows and columns of the coefficient matrix $\mathbf{S}$ are individually scaled by the diagonal matrix $\mathbf{D} = diag[A_1/r_1, \ldots, A_N/r_N]$. Hence, the CG algorithm is applied to solve the following linear system of equations

$$\tilde{\mathbf{S}}\tilde{\mathbf{b}} = \tilde{\mathbf{e}} \tag{6}$$

where $\tilde{\mathbf{S}} = \mathbf{D}^{-1/2}\mathbf{S}\mathbf{D}^{-1/2} = \mathbf{D}^{-1/2}\mathbf{D}\mathbf{C}\mathbf{D}^{-1/2} = \mathbf{D}^{1/2}\mathbf{C}\mathbf{D}^{-1/2}$ has unit diagonals, $\tilde{\mathbf{b}} = \mathbf{D}^{1/2}\mathbf{b}$, and $\tilde{\mathbf{e}} = \mathbf{D}^{-1/2}\mathbf{D}\mathbf{e} = \mathbf{D}^{1/2}\mathbf{e}$. Thus, the vector $\mathbf{D}\mathbf{e}$ on the right-hand side in Eq. 5 is also scaled, and $\tilde{\mathbf{b}}$ must be scaled back at the end to obtain the original solution vector $\mathbf{b}$ (i.e., $\mathbf{b} = \mathbf{D}^{-1/2}\tilde{\mathbf{b}}$). The eigenvalues of the scaled coefficient matrix $\tilde{\mathbf{S}}$ (in Eq. 6) are more likely to be grouped together than those of the unscaled matrix $\mathbf{S}$ (in Eq. 5), thus resulting in a better condition number.

The entries of the scaled coefficient matrix $\tilde{\mathbf{S}}$ are of the following structure:

$$\tilde{s}_{ij} = \begin{cases} -\sqrt{d_{ii}}c_{ij}\dfrac{1}{\sqrt{d_{jj}}} = -\sqrt{\dfrac{A_i}{r_i}}\, r_i F_{ij} \sqrt{\dfrac{r_j}{A_j}} \\ \quad = -\sqrt{r_i A_i}\sqrt{\dfrac{r_j}{A_j}}\, F_{ij} \quad \text{if } i \neq j \\ 1 \qquad \text{otherwise.} \end{cases}$$

The values of the scaling parameters $\sqrt{r_i A_i}$ and $\sqrt{r_j/A_j}$ depend only on the area and reflectivity values of the patches and do not change throughout the iterations. Therefore, the values of the scaling parameters can be computed once at the beginning of the solution phase and maintained in two vectors (for each color band) representing two diagonal matrices $\mathbf{D}_1 = diag[\sqrt{r_1 A_1}, \ldots, \sqrt{r_N A_N}]$ and $\mathbf{D}_2 = diag[\sqrt{r_1/A_1}, \ldots, \sqrt{r_N/A_N}]$. The basic steps of the SCG algorithm proposed for the solution phase of the radiosity method is illustrated in Fig. 2. The $\mathbf{p}^k$ and $\tilde{\mathbf{r}}^k$-vectors in Fig. 2 denote the direction and residual vectors at iteration k, respectively. Note that $\tilde{\mathbf{r}}^k = \tilde{\mathbf{e}} - \tilde{\mathbf{S}}\tilde{\mathbf{b}}^k$ must be null when $\tilde{\mathbf{b}}^k$ is coincident with the solution vector.

Initially, choose $\tilde{\mathbf{b}}^0$ and let $\tilde{\mathbf{r}}^0 = \tilde{\mathbf{e}} - \tilde{\mathbf{S}}\tilde{\mathbf{b}}^0$ and then compute $< \tilde{\mathbf{r}}^0, \tilde{\mathbf{r}}^0 >$

for k = 0, 1, 2, ...

1. form $\mathbf{q}^k = \tilde{\mathbf{S}}\mathbf{p}^k$ as
   $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$; $\mathbf{y} = \mathbf{F}\mathbf{x}$; $\mathbf{z} = \mathbf{D}_1\mathbf{y}$; $\mathbf{q}^k = \mathbf{p}^k - \mathbf{z}$
2. (a) $\theta = < \mathbf{p}^k, \mathbf{q}^k >$
   (b) $\alpha = \dfrac{<\tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k>}{\theta}$
3. $\tilde{\mathbf{r}}^{k+1} = \tilde{\mathbf{r}}^k - \alpha\mathbf{q}^k$
4. $\tilde{\mathbf{b}}^{k+1} = \tilde{\mathbf{b}}^k + \alpha\mathbf{p}^k$
5. (a) $\gamma = < \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} >$
   (b) $\beta = \dfrac{\gamma}{<\tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k>}$
   (c) $\mathbf{r}^k \leftarrow \mathbf{D}_2\tilde{\mathbf{r}}^k$, $\mathbf{b}^k \leftarrow \mathbf{D}_2\tilde{\mathbf{b}}^k$
   check $\mathrm{Norm}(\mathbf{r}^k)/max(\mathbf{b}^k) < \epsilon$
   (d) $< \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} >= \gamma$
6. $\mathbf{p}^{k+1} = \tilde{\mathbf{r}}^{k+1} + \beta\mathbf{p}^k$

Fig. 2. Basic steps of the SCG method

The matrix vector product $\mathbf{q}^k = \tilde{\mathbf{S}}\mathbf{p}^k$ looks as if the $\tilde{\mathbf{S}}$ matrix is to be computed and stored for each color band. However, this matrix vector product can be rewritten for each color as:

$$\begin{aligned} \mathbf{q}^k(r,g,b) &= \tilde{\mathbf{S}}(r,g,b)\mathbf{p}^k(r,g,b) \\ &= [\mathbf{I} - \mathbf{D}_1(r,g,b)\mathbf{F}\mathbf{D}_2(r,g,b)]\mathbf{p}^k(r,g,b) \\ &= \mathbf{p}^k(r,g,b) - \mathbf{D}_1(r,g,b)\mathbf{F}\mathbf{D}_2(r,g,b) \\ &\quad \times \mathbf{p}^k(r,g,b). \end{aligned} \tag{7}$$

Hence, it suffices to compute and store only the original $\mathbf{F}$ matrix and two scaling vectors $\mathbf{D}_1$ and $\mathbf{D}_2$ for each color band for the SCG method. However, in order to minimize the computational overhead during iterations due to this storage scheme, the vector $\mathbf{D}_1\mathbf{F}\mathbf{D}_2\mathbf{p}^k$ should be computed as a sequence of three matrix vector products, $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$, $\mathbf{y} = \mathbf{F}\mathbf{x}$ and $\mathbf{z} = \mathbf{D}_1\mathbf{y}$, which take $\Theta(N), \Theta(M)$ and $\Theta(N)$ times, respectively. Since $M = O(N^2)$, the computational overhead due to the diagonal matrix vector products $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$, $\mathbf{z} = \mathbf{D}_1\mathbf{x}$, and the vector subtraction $\mathbf{q}^k = \mathbf{p}^k - \mathbf{z}$ (which also takes $\Theta(N)$ time) is negligible. The

computational complexity of a single SCG iteration is

$$T_{SCG} \approx (2M + 18N)t_{calc}. \quad (8)$$

Although the operations shown convert the **C** matrix into a symmetric matrix, in practice one should be careful when using the SCG method. The hemicube method used in the form-factor calculations is an approximation. As a result, the form-factor values calculated may contain numeric errors due to the violation of some assumptions (Baum et al. 1989). Therefore, the reciprocity relation may not hold, and the operations may still result in a nonsymmetric matrix.

2.2.3 Convergence check

The convergence of iterative methods is usually checked by comparing a selected norm of the residual error vector $\mathbf{r}^k = \mathbf{e} - \mathbf{C}\mathbf{b}^k$ with a predetermined threshold value at each iteration k. In this work, the following error norm is used for the convergence check:

$$\text{Error}^k = \frac{\sum_{i=1}^{N} |r_i^k|}{max(|b_i^k|)}, \quad (9)$$

where $|\cdot|$ denotes the absolute value. Iterations are terminated when the error becomes less than a predetermined threshold value (e.g., $error^k < \varepsilon$ where $\varepsilon = 5 \times 10^{-6}$). Note that the residual vector $\mathbf{r}^k$ is already computed n the SCG method. However, the residual vector $\mathbf{r}^k$ is not explicitly computed in the GJ scheme. Nonetheless,

$$\mathbf{r}^k = \mathbf{e} - \mathbf{C}\mathbf{b}^k = \mathbf{e} - (\mathbf{I} - \mathbf{RF})\mathbf{b}^k$$

$$= \mathbf{e} + \mathbf{RF}\mathbf{b}^k - \mathbf{b}^k = \mathbf{b}^{k+1} - \mathbf{b}^k. \quad (10)$$

Hence, the residual error vector $\mathbf{r}^k$ can easily be calculated at each iteration of the GJ scheme by a single vector subtraction operation.

3 Related work

There are various parallel implementations for progressive refinement and gathering methods in the literature (Chalmers and Paddon 1989, 1990, 1991; Price and Truman 1990; Purgathofer and Zeiller 1990; Feda and Purgathofer 1991; Guitton et al. 1991; Jessel et al. 1991; Drucker and Schroeder 1992; Varshney and Prins 1992; Paddon et al. 1993; Aykanat et al. 1996). In this section, parallel approaches for the gathering method are summarized.

Price and Truman (1990) parallelize the gathering method on a transputer-based architecture in which the processors were organized as a ring having a master processor, used for communicating with host and graphics system, and had a number of slave processors to do the calculations. Any data can be exchanged with this ring interconnection. In their approach, they assume that total scene data can be replicated in the local memories of the processors, hence form factors can be computed without any interprocessor communication. The GJ iterative scheme is used in their solution phase.

Purgathofer and Zeiller (1990) use a ring of transputers. In the form-factor computation phase, "receiving" patches are statically distributed to worker processors. Patches are distributed to processors randomly to obtain a better load balance. The master processor sends global patch information in blocks to the first processor in the ring. Then, the patch information is circulated in the ring. In their approach, the sparsity of the form-factor matrix is exploited, and the matrix is maintained in compressed form. The memory used for matrix rows and hemicube information is overlapped, allowing the calculation of several rows of the matrix at a time in each processor. The number of rows calculated at a time decreases as more rows allocate memory shared with hemicube information.

Chalmers and Paddon (1989, 1990) use a demand-driven approach in the form-factor computation phase and a data-driven approach in the solution phase in which data are assigned to processors in a static manner. The target architecture is based on transputers arranged in a structure of minimal path lengths. Paddon et al. (1993) discuss the trade-offs between demand and data-driven schemes in the parallelization of the form-factor computation phase. Chalmers and Paddon (1989) address the data redistribution issue for better load balancing in the solution phase. Chalmers and Paddon (1990) use a demand-driven

approach for the form-factor calculation phase. The form-factor row computations are conceptually divided evenly among the processors. The even decomposition here refers to the equal number of row allocations to each conceptual region. Each processor is assigned a task by the master from its conceptual region until all tasks in its region are consumed. Idle processors whose conceptual regions are totally consumed are assigned tasks from the conceptual regions of other processors. However, in such cases, the computed form-factor vectors are passed to the processors that own the conceptual region. The GJ iterative scheme is used in the solution phases of all these works.

4 The hypercube multicomputer

Among the many interconnection topologies, the hypercube topology is popular because many other topologies, such as ring and mesh, can be embedded into it. Therefore, it is possible to arrange processors in the most suitable interconnection topology for the solution of the problem. A d-dimensional hypercube consists of $P = 2^d$ processors (nodes) with a link between every pair of processors whose d-bit binary labels differ in one bit. Thus, each processor is connected to d other processors. The *hamming distance* between two processors in a hypercube is defined as the number of different bits between these two processors' ids. The *channel i* refers to the communication links between processors whose processor ids differ in only the ith bit.

Intel's iPSC/2 hypercube is a distributed memory multicomputer. Data are exchanged and synchronized between nodes via exchanging messages. In this multicomputer, interprocessor communication time (T_{comm}) for transmitting m words can be modeled as $T_{comm} = t_{su} + mt_{tr}$, where t_{tr} is the transmission time per word, and t_{su} ($\gg t_{tr}$ in the iPSC/2 hypercube multicomputer) is the set-up time.

5 Computation of the parallel form-factor matrix

In this section, the parallel algorithms devised for the phase computing the form-factor matrix in the gathering method are described.

5.1 Static assignment

In this scheme, each processor is statically assigned the responsibility of computing the rows corresponding to a subset of patches prior to the parallel execution of this phase. However, projection computations onto local hemicubes may introduce load imbalance during the parallel form-factor computation phase. The complexity of the projection of an individual patch onto a hemicube depends on several geometric factors. A patch that is clipped completely requires much less computation than a visible patch, since it leaves the projection pipeline in a very early stage. Furthermore, a patch with a large projection area on a hemicube requires more scan-conversion computation than a patch with a small projection area. Hence, the assignment scheme should be carefully selected in order to maintain the load balance in this phase.

In this work, we recommend two types of static assignment schemes: *scattered* and *random*. In the scattered assignment scheme, the adjacent patches on each surface are ordered consecutively. Then, the successive patches in the sequence are assigned to the processors in a round-robin fashion. Note that filling the hemicube for the adjacent patches is expected to take an almost equal amount of computation due to the similar view volumes of adjacent patches. Hence, scattered assignment is expected to yield a good load balance. The scattered assignment of the patches on a regular surface (e.g., rectangular surface) is trivial. Unfortunately, this assignment scheme may necessitate expensive preprocessing computations for the irregular surfaces. The random assignment scheme is recommended if the scene data are not suitable for the preprocessing needed for the scattered assignment scheme. In this assignment scheme, randomly selected patches are similarly assigned to the processor in a round-robin fashion. It has been observed experimentally that the random assignment scheme yields a fairly good load balance for a sufficiently large $\frac{N}{P}$ ratio. The random assignment scheme is used in this work.

In both of these two assignment schemes, first ($N \bmod P$) processors in the decimal processor ordering are assigned $\lceil \frac{N}{P} \rceil$ patches, whereas the remaining processors are assigned $\lfloor \frac{N}{P} \rfloor$ patches. After the assignment, the patches are renumbered

so that $\frac{N}{P}$ patches assigned to processor l are renumbered from $\frac{N}{P}l$ to $\frac{N}{P}(l+1)-1$. The new global numbering (new patch ids) is not modified throughout the computations.

5.1.1 Patch circulation

In this scheme, the host processor distributes only local path information to node processors. After receiving the local patch information, each processor calculates the rows of the **F** matrix for its local patches. Each processor places a hemicube around the center of a local patch and calculates the form-factor row for that patch. Each processor's local patch information is circulated among the processors so that it can project all patches to their local hemicubes. The ring-embedded hypercube structure, which can easily be achieved by *gray-code* ordering (Ranka and Sahni 1990), is used for patch circulation. If the number of patches N is not a multiple of P, those processors having $\lceil \frac{N}{P} \rceil$ local patches require one more patch circulation phase than the processors having $\lfloor \frac{N}{P} \rfloor$ local patches. Hence, those processors having $\lfloor \frac{N}{P} \rfloor$ patches participate in an extra patch circulation phase (which does not include any local hemicube fill operation) for the sake of other processors.

In this scheme, information for $\Theta(\frac{N}{P})$ patches is concurrently transmitted to successive processors of the ring in each communication step. The total volume of concurrent communication in a single circulation step is then $\Theta(\frac{N}{P})(P-1) = \Theta(N)$. Hence, the total concurrent communication volume is $\Theta(N)\lceil \frac{N}{P} \rceil = \Theta(\frac{N^2}{P})$. This communication overhead can be reduced by avoiding communication as much as possible. To do this, the global patch information is duplicated at each node processor. The scheme to implement this idea is given in the following section.

5.1.2 Storage sharing scheme

Dynamic memory allocation is used for storing the computed **F** matrix rows. This scheme can be exploited to share the memory needed for global patch information with the memory to be allocated to nonzero matrix elements. With such a sharing of memory, we can avoid interprocessor communication until the memory allocated to global patch information is required for a row of the matrix.

In this scheme, the global patch information is duplicated in each processor after the local patch assignment and the corresponding global patch renumbering mentioned earlier. Then, processors concurrently compute and store the form-factor rows corresponding to their local assignment. They avoid interprocessor communication until no more memory can be allocated for the new row. If a processor cannot allocate memory space for storing the computed form-factor row, it broadcasts a message so that other processors can switch to the communication phase as soon as possible and run the *patch circulation* scheme. Note that the patch should be circulated until all remaining rows of the **F** matrix are calculated. Therefore, the data circulation phase in the patch circulation scheme should be repeated a number of times equal to the maximum number of unprocessed patches remaining.

5.2 Demand-driven assignment

This approach is an attempt to achieve better load balance through patch assignment to idle processors upon request. The scheme proposed in this work has similarities to the approach used by Chalmers and Paddon 1989, (1990). However, unlike their scheme, the patches are not divided *conceptually*. When a patch is processed, the computed form-factor row remains in the processor. In this scheme, each node processor demands a new patch assignment from the host processor as soon as it computes the form-factor row(s) associated with the previous patch assignment. The host processor sends the necessary information for a predefined number of patch assignments to the requesting node processor. The number of patch assignments at a time is a factor that affects the performance. The number of node-to-host and host-to-node communications decreases with an increasing number of patch assignments at a time. However, this may affect the quality of load balance adversely.

Each node processor keeps an array to save the reflectivity and emission values of the processed patches. The global ids of the processed patches are also saved in an array to be used in the

solution phase. In addition, each processor holds the information for all patches in the scene to avoid interprocessor communication, as is explained in Sect. 5.1.2. The host processor behaves as a master. It is responsible for processing requests and synchronizing nodes between phases. The host program maintains an array for global patch information and keeps account of the remaining patches to be processed. All node processors are synchronized by the host processor when one or more of the node memories become full, and processors have to switch to the data circulation mode. The host is also responsible for the termination of the form-factor computation phase.

6 Parallel solution phase

This section describes the parallel GJ and SCG algorithms developed for the solution phase. The parallel implementation of the solution phase is closely related to the schemes used in the phase computing the form-factor matrix because patch distribution, hence row distribution, to the processors differs in each scheme. In the following sections, parallel algorithms for the solution phase are described. We assume that a static assignment scheme is used in the phase computing the parallel form-factor. An efficient parallel renumbering scheme, described in Sect. 6.3, adapts these algorithms if the demand-driven assignment scheme is used in this phase.

6.1 Parallel Gauss–Jacobi (GJ) method

The GJ algorithm formulated (Fig. 1) for the solution phase has the following basic types of operations: matrix vector product ($\mathbf{x} = \mathbf{F}\mathbf{b}^k$), diagonal matrix vector product ($\mathbf{y} = \mathbf{R}\mathbf{x}$), vector subtraction/addition ($\mathbf{b}^{k+1} = \mathbf{y} + \mathbf{e}$, $\mathbf{r}^k = \mathbf{b}^{k+1} - \mathbf{b}^k$), vector norm and maximum (step 3). All of these basic operations can be performed concurrently by distributing the rows of the form-factor matrix **F**, the corresponding diagonals of the **R** matrix, and the corresponding entries of the **b** and **e** vectors. In the parallel form-factor computation, each processor computes the complete row of form factors for its local patches. Hence, each processor holds a row slice of the form-factor matrix at the end of the form-factor computation phase. Thus, the row

Initially, choose $\mathbf{b}^0$
for k = 1,2,3,...

1. (a) perform global-concatenate on $\mathbf{b}^k_{local} \rightarrow \mathbf{b}^k_{global}$
   (b) $\mathbf{x}_{local} = \mathbf{F}_{local}\mathbf{b}^k_{global}$
   (c) $\mathbf{y}_{local} = \mathbf{R}_{local}\mathbf{x}_{local}$
   (d) $\mathbf{b}^{k+1}_{local} = \mathbf{y}_{local} + \mathbf{e}_{local}$
2. $\mathbf{r}^k_{local} = \mathbf{b}^{k+1}_{local} - \mathbf{b}^k_{local}$
3. (a) $\sigma_{local} = \mathrm{Norm}(\mathbf{r}^k_{local})$
   $b_{localmax} = max(\mathbf{b}^k_{local})$
   (b) perform one global-sum-max operation to compute
   $\sigma_{local} \rightarrow \sigma_{global}$ and $b_{localmax} \rightarrow b_{globalmax}$
   check $\sigma_{global}/b_{globalmax} < \epsilon$

Fig. 3. The parallel GJ algorithm

partitioning required for the parallelization of the solution phase is automatically achieved in this phase. The slices of the **R**, **b**, and **e** vectors are mapped to the processors accordingly. With such a mapping, each processor is responsible for updating the values of those **b** and **r** vector elements assigned to it. That is, each processor is responsible for updating its own slice of the global **b** vector in a local **B** array (of size N/P) at each iteration. Figure 3 illustrates the parallel GJ algorithm.

In an individual GJ iteration, each processor needs to calculate a local matrix vector product that involves $\frac{N}{P}$ inner products of its local rows with the global **b** vector. To do this, the whole **b** vector computed in a distributed manner in a particular iteration is needed by all processors in the next iteration. This requirement necessitates the *global concatenate* operation, which is illustrated for a 3D hypercube topology in Fig. 4. In this operation, each processor l moves its local **b** array to the lth slice of a working array GB of size N. Then, $log_2 P$ concurrent exchange communication steps are taken between neighbor processors over channels $j = 0, 1, 2, \ldots, log_2 P - 1$, as is illustrated in Fig. 4. Note that the amount of concurrent data exchange between processors is only $\frac{N}{P}$ in the first step, and it is doubled at each successive step. That is, at the ith communication step, processors exchange the appropriate slices of size $2^{(i-1)}\frac{N}{P}$ of their local GB array over channel $i - 1$. Therefore, the total volume of concurrent communication is:

$$\text{Volume of communication} = \sum_{j=0}^{log_2 P - 1} \frac{2^j N}{P}$$

$$= \frac{(P-1)N}{P} \text{ words.} \qquad (11)$$

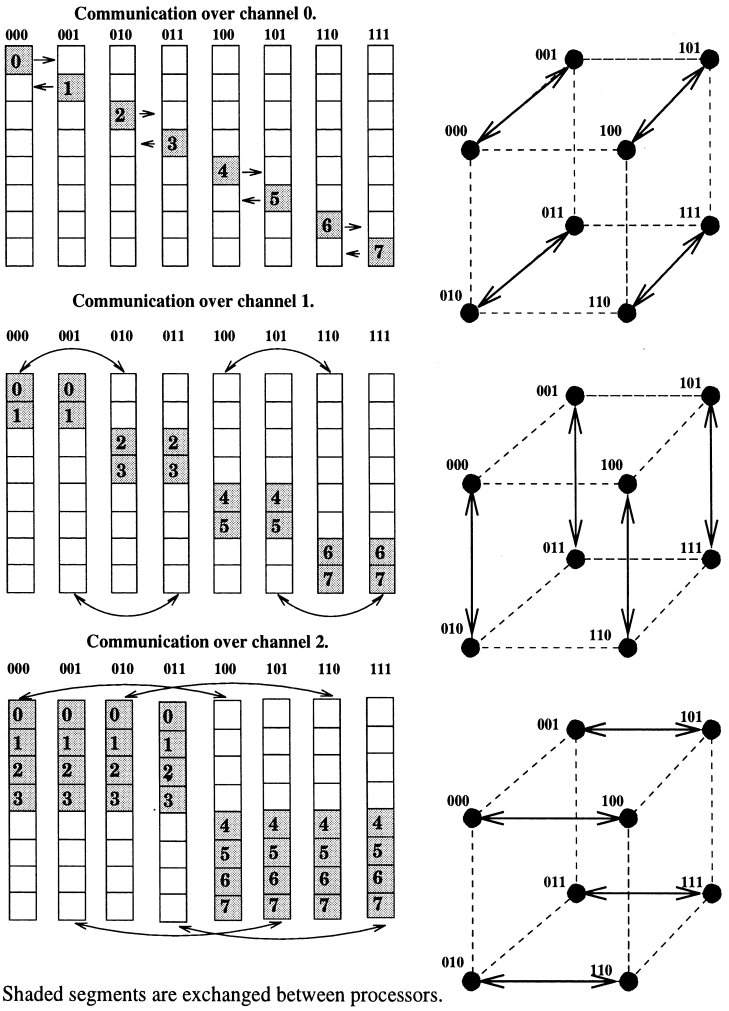


Fig. 4. Global concatenate operation for a 3D hypercube

The distributed vector add/subtract operations are performed concurrently as local vector operations on vectors of size $\frac{N}{P}$ without necessitating any interprocessor communication. The partial sums computed by each processor must be added to form the global sum to compute the vector norm ($\sum_{i=1}^{N} |r_i|$). In addition, local $b_{localmax}$ values should be compared to obtain the global maximum ($b_{globalmax}$). Furthermore, the results should be distributed to all processors in order to ensure the termination at the same iteration. The distributed global norm operation can be performed by a *global sum* operation. The exchange-communication sequence of the global sum operation is exactly the same as that of global concatenate operation. The only difference is the local scalar addition after each exchange communication step (Aykanat et al. 1988) instead of the local vector concatenation. This local scalar operation involves the addition of the received partial sum to the current partial sum. Similarly, the distributed global maximum can be found by using the *global max* operation. The global max operation can be done by replacing the local scalar addition operation with a comparison operation in global sum operation. Performing global sum and global max operations successively requires $2log_2P$ of set-up time. Fortunately, this set-up time can be decreased to log_2P by combining the global max and global sum operations into a single global operation (global sum-max). In this global operation, partial sums and current maximums

are exchanged after each exchange communication step. Therefore, assuming a perfect load balance, the parallel computational complexity of an individual GJ iteration is:

$$T_{GJ} \approx \left(\frac{2M}{P} + \frac{6N}{P}\right) t_{calc} + 2log_2 P t_{su} + \left(\frac{P-1}{P} N + 2log_2 P\right) t_{tr}. \quad (12)$$

As is seen from this equation, the communication overhead can be considered negligible for a sufficiently large granularity (i.e., $M/P \gg N$). Note that this equation is equivalent to Eq. 4 for $P = 1$.

6.2 Parallel scaled conjugate-gradient (SCG) method

The SCG algorithm (Fig. 2) formulated for the solution phase has the following basic types of operations: matrix-vector product ($\mathbf{y} = \mathbf{F}\mathbf{x}$), diagonal matrix-vector products ($\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$, $\mathbf{z} = \mathbf{D}_1\mathbf{y}$), vector subtraction ($\mathbf{q}^k = \mathbf{p}^k - \mathbf{z}$), inner products ($\langle \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k \rangle$, $\langle \mathbf{p}^k, \mathbf{q}^k \rangle$), vector update through SAXPY operations in steps 3, 4, and 6, and vector norm and maximum operations for the convergence check. All of these basic operations can be performed concurrently by distributing the rows of the $\mathbf{F}$ matrix and the corresponding entries of the $\mathbf{e}, \mathbf{D}_1, \mathbf{D}_2, \mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{b}, \mathbf{p}, \tilde{\mathbf{r}}$, and $\mathbf{q}$ vectors.

As is discussed for the parallel GJ algorithm, during the phase of the parallel form-factor computation, the rows of the $\mathbf{F}$ matrix are assigned to the processors automatically. With such a mapping, each processor stores its own (local) row slice of the $\mathbf{F}$ matrix and the corresponding slices of the $\mathbf{e}, \mathbf{D}_1$, and $\mathbf{D}_2$ vectors. Furthermore, each processor is responsible for updating its local slices of the $\mathbf{x}, \mathbf{y}, \mathbf{z}, \mathbf{b}, \mathbf{p}, \tilde{\mathbf{r}}, \mathbf{q}$ vectors. Figure 5 illustrates the parallel SCG algorithm.

A sequence of distributed matrix and vector computations are needed for the distributed computation of the $\mathbf{q}^k$ vector. To find the diagonal matrix vector products $\mathbf{x} = \mathbf{D}_2\mathbf{p}^k$ and $\mathbf{z} = \mathbf{D}_1\mathbf{y}$, the processors concurrently compute their local $\mathbf{x}_{local}$ and $\mathbf{z}_{local}$ vectors by computing element-by-element products of pairs of local vectors that correspond

for k = 0, 1, 2, ...

1. (a) $\mathbf{x}_{local} = \mathbf{D}_{2(local)}\mathbf{p}^k_{local}$
   (b) perform global-concatenate on $\mathbf{x}_{local} \rightarrow \mathbf{x}_{global}$
   (c) $\mathbf{y}_{local} = \mathbf{F}_{local}\mathbf{x}_{global}$
   (d) $\mathbf{z}_{local} = \mathbf{D}_{1(local)}\mathbf{y}_{local}$
   (e) $\mathbf{q}^k_{local} = \mathbf{p}^k_{local} - \mathbf{z}_{local}$
2. (a) $\theta_{local} = < \mathbf{p}^k_{local}, \mathbf{q}^k_{local} >$
   (b) perform global-sum on $\theta_{local} \rightarrow \theta_{global}$
   (c) $\alpha_{global} = < \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k >_{global} / \theta_{global}$
3. $\tilde{\mathbf{r}}^{k+1}_{local} = \tilde{\mathbf{r}}^k_{local} - \alpha_{global}\mathbf{q}^k_{local}$
4. $\tilde{\mathbf{b}}^{k+1}_{local} = \tilde{\mathbf{b}}^k_{local} + \alpha_{global}\mathbf{p}^k_{local}$
5. (a) $\lambda_{local} = < \tilde{\mathbf{r}}^{k+1}_{local}, \tilde{\mathbf{r}}^{k+1}_{local} >$
   (b) $\mathbf{r}^k_{local} \leftarrow \mathbf{D}_2\tilde{\mathbf{r}}^k_{local}$, $\mathbf{b}^k_{local} \leftarrow \mathbf{D}_2\tilde{\mathbf{b}}^k_{local}$
   $b_{localmax} = max(\mathbf{b}^k_{local})$, $\sigma_{local} = \mathrm{Norm}(\mathbf{r}^k_{local})$
   (c) perform one global operation to compute
   $\lambda_{local} \rightarrow \lambda_{global}$, $\sigma_{local} \rightarrow \sigma_{global}$, $b_{localmax} \rightarrow b_{globalmax}$
   check $\sigma_{global} / b_{globalmax} < \epsilon$
   (d) $\beta_{global} = \lambda_{global} / < \tilde{\mathbf{r}}^k, \tilde{\mathbf{r}}^k >_{global}$
   (e) $< \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} >_{global} = \lambda_{global}$
6. $\mathbf{p}^{k+1}_{local} = \tilde{\mathbf{r}}^{k+1}_{local} + \beta_{global}\mathbf{p}^k_{local}$

Fig. 5. The parallel SCG method

to their slices of the global $\mathbf{D}_2$, $\mathbf{p}^k$ and $\mathbf{D}_1$, $\mathbf{y}$ vectors, respectively. Thus, the distributed diagonal matrix vector product does not necessitate any interprocessor communication. As is the case in the GJ method, the distributed computation of the matrix vector product $\mathbf{y} = \mathbf{F}\mathbf{x}$ necessitates global concatenation on the local $\mathbf{x}_{local}$ vector stored in a local array X of size N/P in each processor. Then, the processors concurrently compute their local $\mathbf{y}_{local}$ vectors by multiplying a local matrix corresponding to their slice of the $\mathbf{F}$ matrix with the global $\mathbf{x}_{global}$ vector, collected in an array GX of size N in their local memories after the global concatenate operation. Finally, the processors concurrently compute their local $\mathbf{q}^k_{local}$ vectors with local vector subtraction operation.

All processors need the most recently updated values for the global scalars α_{global} and β_{global} for their local vector updates in steps 3, 4, and 6. As is seen in steps 2 and 5, the update of these global scalars involves computing the inner products $\langle \mathbf{q}^k, \mathbf{p}^k \rangle$ and $\langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle$ at each iteration. Hence, all processors should receive the results of these distributed inner product computations. To compute these products, the processors concurrently compute the local inner products (partial sums) corresponding to their slices of the respective global vectors. Then, the results are accumulated in the local memory of each processor by a global-sum operation. At the end of the global sum

operation, the processors can concurrently compute the same value for the global scalars α_{global} and β_{global} with these global inner product results. In steps 3, 4, and 6, the processors concurrently update their local $\mathbf{b}_{local}$, $\tilde{\mathbf{r}}_{local}$ and $\mathbf{p}_{local}$ vectors with local SAXPY operations on these vectors. Note that the distributed norm and maximum operations also necessitate global sum-max operations after all processors concurrently compute their local (partial) error norms and maximums, which correspond to the norm/maximum of their slices of the global **r** and **b** vectors. Fortunately, these operations and the global inner product $\langle \tilde{\mathbf{r}}^{k+1}, \tilde{\mathbf{r}}^{k+1} \rangle$ can be concurrently accumulated in the same global operation to avoid the extra overhead of the $log_2 P$ set-up time. In this global operation, one local maximum and two partial sums are exchanged in each of the $log_2 P$ concurrent exchange steps. Therefore, assuming a perfect load balance, the parallel computational complexity of an individual SCG iteration is:

$$T_{SCG} \approx \left(\frac{2M}{P} + \frac{18N}{P}\right) t_{calc} + 3 log_2 P t_{su} + \left(\frac{P-1}{P} N + 4 log_2 P\right) t_{tr}. \quad (13)$$

As is seen from this equation, the communication overhead can be considered negligible for a sufficiently large granularity (i.e., $M/P \gg N$).

6.3 A parallel renumbering scheme

In the form-factor computation phase of the static assignment scheme, patches are renumbered and assigned to the node processors so that processor l has patches from $\frac{N}{P} l$ to $\frac{N}{P}(l+1) - 1$ for $l = 0, 1, 2, \ldots, P-1$. Therefore, the exchange sequence, together with the local concatenate scheme in the global concatenate operations at step 1a of the parallel GJ algorithm and step 1b of the parallel SCG algorithm, maintains the original global patch ordering of the static assignment scheme in the local copies of the global vectors. Hence, during the concurrent local matrix vector products, the appropriate entries in the current global vectors collected in the local *GB* (in GJ) and *GX* (in SCG) arrays can be accessed for multiplication by indexing through the *column ids* of the local nonzero form-factor values. Unfortunately, this nice consistency between the global patch numbering and the global **F** matrix row ordering among the processors is disturbed in the demand-driven assignment scheme. Hence, the demand-driven assignment scheme necessitates two-level indexing for each scalar multiplication involved in the local matrix vector products at each iteration. We propose an efficient parallel renumbering scheme to avoid this two-level indexing.

During the computation of the form-factor matrix in the demand-driven assignment scheme, each processor saves the global id of the patches it receives from the host processor in a local integer array *ID*. After the form-factor computation, a global-concatenate operation on these local arrays collects a copy of the global *GID* array in each processor. Note that the collection operation is done in the same way as the global concatenate operation to be performed on the local *B* (in GJ) and *X* (in SCG) arrays during the iterations. Hence, there is a one-to-one correspondence between the *GB* (*GX*) and *GID* arrays such that the radiosity value in *GB*[i] (*GX*[i]) belongs to the patch whose original global id = *GID*[i]. Then, each processor constructs the same permutation array *PERM* of size *N*, where *PERM*[*GID*[i]] = i (for i = 1, 2, ..., *N*) by performing a single *for* loop. Here, *PERM*[i] denotes the new global id for the *i*th patch in the original global numbering. Then, each processor concurrently updates the *column_id* values of all its local nonzero form-factor values using the *PERM* array as *column_id* = *PERM*[*column_id*]. Note that this renumbering operation is performed only once as a preprocessing step just before the solution phase, and it is not repeated when the reflectivity and/or emission values are modified.

7 Load balancing in the solution phase: data redistribution

Assigning equal numbers of **F** matrix rows and the corresponding vector elements suffices to achieve a perfect load balance during the distributed vector operations involved in the GJ and SCG iterations. However, the computational

complexities of individual GJ and SCG iterations are bounded by the distributed matrix vector products **Fb** and **Fx**, respectively. Hence, the load balance during the computing of the distributed matrix vector products is much more crucial than during that of the vector operations. Since we exploit the sparsity of the **F** matrix in the matrix vector products, the load balance in these computations can only be achieved by assigning equal numbers of nonzero entries of the **F** matrix to the processors.

The factors that effect the load balance in the form-factor computation and the solution phases are not the same. The assignment schemes mentioned earlier for the form-factor computation aim at achieving load balance on the hemicube filling operations associated with the patches. However, even if two patches require almost equal time for hemicube filling the number of nonzero entries in the respective rows may be substantially different. Hence, an assignment scheme (e.g., the demand-driven assignment) that yields a near-perfect load balance in the form-factor computation may not achieve a good load balance during the solution phase. Furthermore, it is not possible to achieve perfect load balance in the form-factor computation phase through static assignment, since the amount of projection work is not known a priori. However, once the sparsity structure of the **F** matrix is determined at the end of the form-factor computation phase, static reassignment can be used for better load balancing in the solution phase. Recall that the parallel form-factor computation phase already imposes a row-wise distribution of the nonzero **F** matrix entries that may not achieve this desired assignment. Hence, a redistribution of **F** matrix entries is needed for perfect load balancing during the distributed matrix vector product operations. There are various data redistribution schemes in the literature (Ryu and Jájá 1990; Jájá and Ryu 1992). The main objective in these schemes is to achieve a data redistribution such that the number of data elements in different processors differ at most by one. However, these schemes do not assume any hierarchy among the data elements. In our case, data elements belong to the rows of the **F** matrix, and it is desirable to minimize the subdivision of rows among the processors because subdivided rows may require extra communication during the solution phase. Furthermore, the data movement necessitated by the redistribution should be minimized to minimize the preprocessing overhead for the solution phase.

7.1 A parallel data-redistribution scheme

In this work, we propose an efficient parallel redistribution scheme that allows at most one shared row between successive processors in the decimal ordering (i.e., between processors l and $l+1$ for $l = 0,1, \ldots, P-2$). That is, each processor l, except the first and the last processors (0 and $P-1$, respectively), may share at most two rows, one with processor $l-1$ and one with processor $l+1$. The processors 0 and $P-1$ may share at most one row with processors 1 and $P-2$, respectively. Recall that successive processors in the decimal ordering hold the successive row slices of the distributed **F** matrix. We assume a similar global implicit numbering for the nonzero entries of the distributed **F** matrix. Nonzero entries in the same row are assumed to be numbered in the storage order. Nonzero entries in the successive rows are assumed to be numbered successively. Hence, the global numbers of the nonzero entries in processor $l+1$ are assumed to follow those of processor l.

In the parallel reassignment phase, a global concatenate operation is performed on the local **F** matrix nonzero entry counts so that each processor collects a copy of the global integer $OLDMAP$ array of size P. At this stage, $OLDMAP[l]$ denotes the number of nonzero entries computed and stored in processor l for $l = 0, 1, \ldots, P-1$. Then, processors concurrently run the prefix sum operation on their $OLDMAP$ array. After the prefix sum operation, $OLDMAP[l-1]+1 \cdots OLDMAP[l]$ denotes the range of nonzero entries computed and stored in processor l in the assumed ordering. Note that $OLDMAP[P-1] = M$ yields the total number of nonzero entries in the global **F** matrix. Then, all processors concurrently construct the same integer $NEWMAP$ array of size P, where $NEWMAP[l] = \lceil \frac{M}{P} \rceil$ for $l = 0, 1, \ldots, (M \bmod P) - 1$ and $NEWMAP[l] = \lfloor \frac{M}{P} \rfloor$ for $l = (M \bmod P), \ldots, P-1$. At this stage, $NEWMAP[l]$ denotes the number of nonzero entries to be stored in processor l after the data redistribution. Then, the processors concurrently run the prefix sum

operation on their $NEWMAP$ array. Therefore, after the prefix sum operation $NEWMAP[l-1]+1 \cdots NEWMAP[l]$ denotes the range of **F** matrix nonzero entries to be stored in processor l in the assumed ordering after the redistribution. Each processor, knowing the new mapping for their current local nonzero entries, can easily determine its local row subslices to be redistributed and their destination processor(s). Similarly each processor, knowing the old mapping for their expected mapping after the data redistribution, can easily determine the source processor(s) from which it will receive data during the redistribution and the volume of data in each receive operation. However, the sending processors should append the row structure of the data transmitted in front of the messages during the data redistribution phase. Note that consecutive row data are transmitted between processors, and only the first and/or last rows of the transmitted data may be partial row(s). Processors receiving data store them in row structure according to the global row ordering with simple pointer operations.

At the end of the data redistribution phase, the number of nonzero entries stored by different processors may differ at most by one. Thus, perfect load balance is achieved during the distributed sparse matrix vector product calculated at each iteration of both the GJ and the SCG methods. However, shared rows need special attention during these distributed matrix-vector-product operations. Consider a row 'i' (in global row numbering) shared between processors l and $l+1$. Note that this row corresponds to the last and first (partial) local rows of processors l and $l+1$, respectively. During the distributed matrix vector product these two processors accumulate the partial sums that correspond to the inner products of their local portions of the ith row of the **F** matrix with the global right-hand-side vector. These two partial sums should be added to determine the ith entry of the resultant left-hand-side vector. Hence, row sharing necessitates one concurrent interprocessor communication between successive processors after each distributed matrix vector product. In the proposed mapping, the computations associated with the vector entries corresponding to the shared rows between processors l and $l+1$ are assigned to the processor $l+1$ for $l = 0, 1, \ldots, P-2$. Hence, processors concurrently send the partial inner product results corresponding to their last local row (if it is shared) to the next processor in the decimal ordering. Note that only a single floating-point word is transmitted in these communications. Hence, this concurrent shift-and-add scheme for handling shared rows introduces a $t_{su} + t_{tr} + t_{add}$ concurrent communication and addition overhead per iteration of both the GJ and SCG algorithms.

7.2 Avoiding the extra set-up time overhead

We propose an efficient scheme for the GJ method that avoids the extra set-up time overhead by incorporating this extra communication into the global concatenation. In the proposed scheme, the global concatenate operation is performed on the $\mathbf{b}_{local}^{k+1}$ array after step 1d (Fig. 3) instead of on the $\mathbf{b}_{local}^{k}$ array at step 1a. That is, it is actually performed for the next iteration. Note that the first and/or last entries of the $\mathbf{x}_{local}$ array may contain partial results at the end of step 1b due to the row sharing. Processors propagate these partial results to their $\mathbf{b}_{local}^{k+1}$ arrays through step 1c and d. Hence, the first and/or last entries of the $\mathbf{b}_{local}^{k+1}$ array may contain partial results just before the global concatenate operation modified to handle these partial results. The exchange and local concatenate structure of the modified global concatenate operation is exactly the same as that of the conventional one. However, just after the concurrent exchange step over channel j, processors whose jth bit of their processor ids are 1(0) add the last (first) entry of the received array to the first (last) entry of their local array in addition to the proper local concatenate operation if this location contains a partial result. The concurrent addition operation after the exchange step over channel j corrects the partial result corresponding to the shared rows between successive processors of the hamming distance $j+1$ for $j = 0, 1, \ldots, log_2 P - 1$. The proposed modification introduces an overhead of $(t_{tr} + t_{add}) log_2 P$ to each global concatenate operation. Since $t_{su} \gg t_{tr}$ in medium-to-coarse grain parallel architectures (e.g., the iPSC/2), the modified global concatenate scheme performs much better than the single shift-and-add scheme. In the SCG method, a similar approach can be followed to incorporate the extra communication overhead due to the shared rows

into the global inner product operation at step 2a-b in Fig. 5.

8 Experimental results

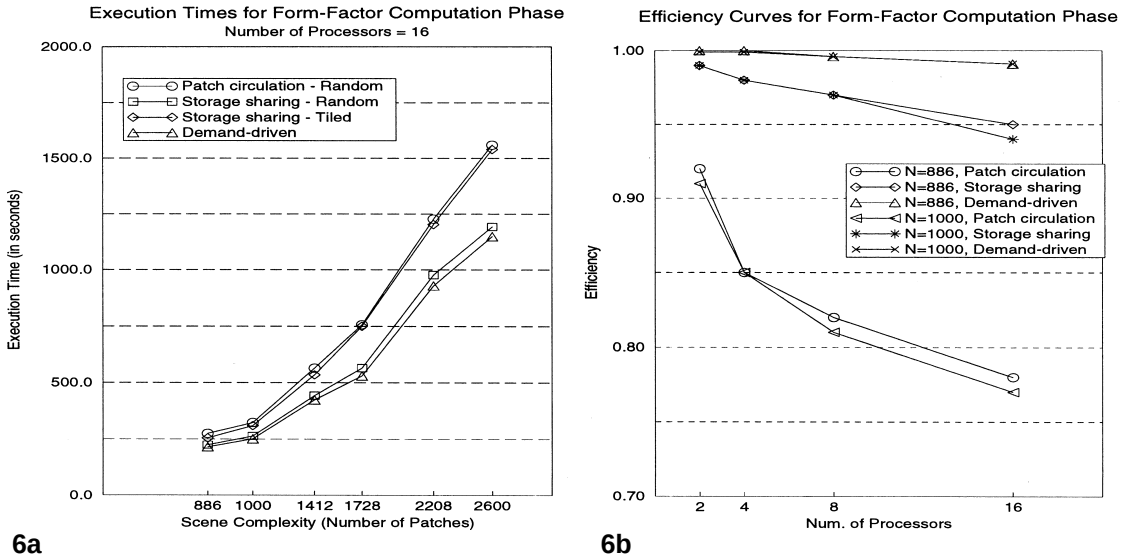
The algorithms discussed in this work were implemented (in the C language) on a 4D Intel iPSC/2 hypercube multicomputer. These algorithms were tested on various room scenes containing various objects discretized into varying numbers of patches ranging from 496 to 2600.

Table 1 illustrates the relative performance results of various parallel algorithms for the form-factor computation phase. The execution times of the algorithms are illustrated in Fig. 6a for 16 processors. Parallel timing results for the random assignment scheme denote the average of 5 different executions for different random assignments. As is seen in the table, the storage sharing scheme gives better performance results than the patch circulation scheme. In the storage sharing scheme, random decomposition yields a better load balance than the tiled decomposition, as is expected. However, tiled assignment in the storage sharing scheme yields better results in most of the test instances (e.g., 15 out of 19) than the random assignment in patch circulation due to the decrease in communication overhead. As seen in Table 1 and in Fig. 6a, the demand-driven scheme always performs better than the static assignment scheme due to a better load balance. Note that experimental timing results for some of the instances are missing for a small number of processors due to insufficient local memory sizes. The sequential timings could only be obtained for the smallest scenes with $N = 886$ and $N = 1000$ as 3418.7 s and 3981.3 s, respectively. The efficiency curves for these scenes are illustrated in Fig. 6b. The demand-driven scheme yields almost 0.99 efficiency even for these two small scenes on a hypercube with 16 processors.

We have also experimented on the effect of the *assignment granularity* on the form-factor computation and solution phases for the demand-driven assignment scheme. The results of these experiments are displayed in Fig. 7. The assignment granularity denotes the number of patches

Table 1. Relative performance results in parallel execution times (in seconds) of various parallel algorithms for the form-factor computation phase

Scene		P	Static assignment			Demand-driven scheme
N	M		Patch circulation	Storage sharing		
			Random	Random	Tiled	
2600	1 804 647	16	1560.0	1193.6	1539.4	1149.9
		8	3046.4	2380.7	3024.9	2299.5
2208	1 468 539	16	1227.8	977.5	1203.5	929.1
		8	2383.4	1911.2	2365.5	1857.3
1728	746 779	16	757.6	565.5	751.0	530.2
		8	1450.9	1099.0	1482.3	1059.0
		4	2719.5	2161.5	2909.9	2110.8
1412	461 947	16	564.9	443.6	535.4	423.9
		8	1078.1	867.1	994.0	843.9
		4	2032.6	1700.7	1923.0	1684.9
		2	3764.7	3399.2	3594.3	3365.3
1000	342 003	16	322.8	263.7	309.9	251.2
		8	616.1	512.3	621.8	499.7
		4	1173.8	1012.5	1149.3	996.8
		2	2191.2	2014.4	2223.8	1993.0
886	303 146	16	274.4	224.8	254.8	215.5
		8	519.1	439.0	492.3	428.9
		4	997.2	870.2	955.7	853.9
		2	1858.5	1719.0	1858.3	1708.1



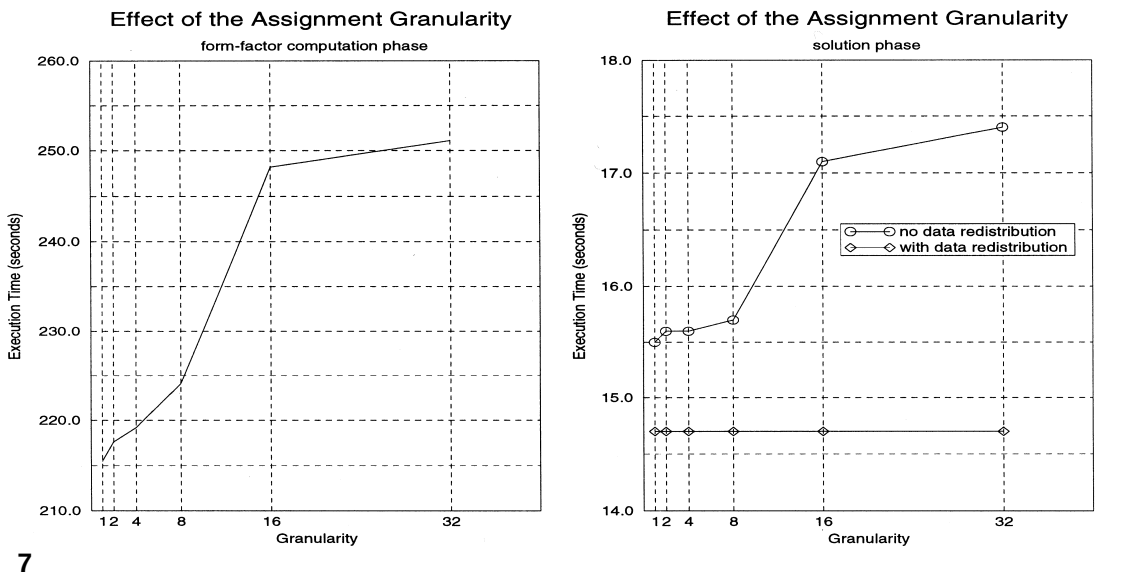


Fig. 6a, b. Form-factor computation phase: **a** execution times for various schemes on 16 processors; **b** efficiency curves for various schemes

Fig. 7. The effect of the assignment granularity on the performance (execution time in seconds) of the demand-driven scheme for $N = 886$, $P = 16$

assigned and sent to the requesting idle processor by the host processor. A small assignment granularity (e.g., single patch assignment) gives better performance in the parallel form-factor computation phase due to the better load balance in spite of the increased communication overhead. Therefore, we can deduce that the calculation of a single form-factor row is computationally intensive, and hence the load balance is a more crucial factor than the communication overhead in this scheme. As is also seen in the figure, a similar behavior is observed in the solution phase when nonzero entries are not redistributed. Note that higher granularity means a processor will generate more rows for a single request. Hence, the number of nonzero entries in the local slices of the

Table 2. Parallel execution times (in seconds) of various schemes in the solution phase (using the GJ method) along with the associated overheads. The total execution time (TOT) including overheads, i.e., TOT = (solution + preprocessing) time

Scene		P	Static assignment, random	Demand-driven approach								
				No redistribution			Redistribution					
				Preprocessing time	Solution time			Preprocessing time		Solution time		
N	M		TOT	Renumbering	Iterations	Total	TOT	Renumbering	Redistribution	Iterations	Total	TOT
1412	461 947	16	28.3	0.149	0.371	28.2	28.3	0.138	0.170	0.346	26.3	26.6
		8	54.2	0.257	0.699	53.1	53.3	0.250	0.117	0.680	51.7	52.1
		4	104.9	0.484	1.368	104.0	104.5	0.477	0.130	1.350	102.6	103.2
		2	207.8	0.935	2.697	205.0	205.9	0.932	0.049	2.691	204.5	205.5
1000	342 003	16	18.2	0.114	0.280	17.9	18.0	0.104	0.093	0.263	16.8	17.0
		8	34.8	0.195	0.523	33.5	33.7	0.189	0.117	0.514	32.9	33.2
		4	66.8	0.364	1.030	65.9	66.3	0.359	0.085	1.020	65.3	65.7
		2	131.3	0.708	2.045	130.9	131.6	0.702	0.024	2.042	130.1	130.9
886	303 146	16	16.0	0.099	0.242	15.5	15.6	0.095	0.074	0.230	14.7	14.9
		8	30.2	0.173	0.463	29.6	29.8	0.167	0.078	0.452	28.9	29.2
		4	58.2	0.320	0.903	57.8	58.2	0.316	0.038	0.894	57.2	57.6
		2	115.3	0.627	1.811	115.9	116.5	0.617	0.030	1.783	114.1	114.8

F matrix in each processor may be substantially different, incurring more load imbalance in the solution phase for higher granularity values. As is expected, when the data are redistributed, the execution time of the solution phase remains constant, irrespective of the assignment granularity.

Table 2 illustrates the performance comparison of various schemes in the solution phase along with the associated overheads. Note that static assignment scheme does not necessitate any renumbering operation. As is expected, data redistribution achieves performance improvement due to better load balancing in spite of the preprocessing overheads. The overall performance gain will be much more notable for repeated solution operations as are required in lighting simulations, since the data are redistributed only once for such applications. As is seen in this table, the time spent for the renumbering and data redistribution operation is substantially smaller than even the solution time per iteration and yields a considerable improvement in performance during the parallel solution. For example, by spending almost 0.6% of the solution time in data redistribution, we reduce the total solution time by almost 7.1% on 16 processors for the scene with $N = 1412$ patches. The relative performance gain achieved by adopting data redistribution is expected to increase with an increasing number of processors. Table 2 also illustrates the decrease in the execution time of the parallel renumbering operation whenever the data redistribution operation is performed. This is due to the fact that the load balance metric in both the parallel renumbering and the matrix vector product operations are exactly the same; i.e., there are equal numbers of nonzero matrix elements in each processor.

Table 3 illustrates the performance comparison of the GJ and SCG methods for the parallel solution phase. Note that experimental timing results for some of the instances on a small number of processors are missing due to the insufficient local memory size. However, sequential timings for the scenes with $N = 1728$, 2208, and 2600 patches are estimated with the sequential complexity expressions given in Eqs. 4 and 8 and with $t_{calc} = 5.87\ \mu$s for the sake of efficiency computations. The number of iterations denote the total number of iterations required for convergence to the same tolerance value (5×10^{-6}) for three color bands (i.e., red, green, blue). As is seen in Table 3, an

Table 3. Performance comparison of parallel the Gauss–Jacobi (GJ) and Scaled Conjugate-Gradient (SCG) methods (1* denotes the estimated sequential timings)

Scene		P	Gauss–Jacobi			Scaled Conjugate-Gradient		
N	M		Execution time		Number of iterations	Execution time		Number of iterations
			Total	Iterations		Total	Iterations	
2600	1 804 647	16	124.0	1.35	92	54.1	1.39	39
		8	246.2	2.68	92	106.7	2.74	39
		1*	1957.6	21.28	92	837	21.47	39
2208	1 468 539	16	102.1	1.10	93	46.4	1.13	41
		8	202.6	2.18	93	91.3	2.23	41
		1*	1610.6	17.32	93	716.4	17.47	41
1728	746 779	16	51.5	0.57	91	23.6	0.59	40
		8	101.6	1.12	91	46.1	1.15	40
		4	202.0	2.22	91	91.0	2.28	40
		1*	803.4	8.83	91	358	8.95	40
1188	178 374	16	12.6	0.15	87	6.1	0.16	38
		8	24.2	0.28	87	11.4	0.30	38
		4	47.4	0.55	87	21.9	0.58	38
		2	93.9	1.08	87	43.1	1.13	38
		1	186.7	2.15	87	83.5	2.20	38
880	45 889	16	4.1	0.05	89	2.4	0.06	41
		8	7.1	0.08	89	4.0	0.10	41
		4	13.3	0.15	89	7.1	0.17	41
		2	26.1	0.29	89	13.6	0.33	41
		1	51.4	0.58	89	25.8	0.63	41
496	66 900	16	4.8	0.06	83	2.5	0.07	38
		8	8.8	0.11	83	4.4	0.12	38
		4	17.2	0.21	83	8.4	0.22	38
		2	33.8	0.41	83	16.4	0.43	38
		1	67.0	0.81	83	31.5	0.83	38

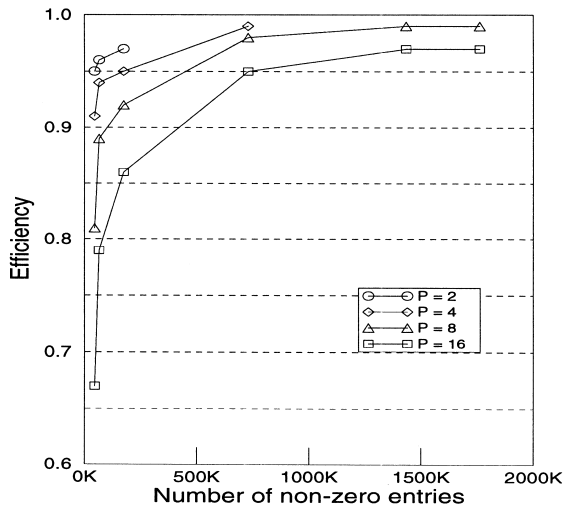


Fig. 8. Efficiency curves for the SCG method

individual SCG iteration takes more time than that of the GJ iteration. However, the SCG method converges much faster than the GJ method, as is expected. Therefore, we recommend the parallel SCG method for the solution phase. Figure 8 illustrates the efficiency curves of the SCG method. As is seen in this figure, the efficiency remains above 86% for sufficient granularity (i.e., $M/P > 11\,148$).

9 Conclusions

In this work, a parallel implementation of the gathering method for hypercube-connected multicomputers has been discussed for applications in which the location of objects and light sources remain fixed, whereas the intensity and color of the light sources and/or reflectivity of objects vary

in time, such as in lighting simulations. In such applications, the efficient parallelization of the solution phase is important since this phase is repeated many times.

The powerful scaled conjugate-gradient (SCG) method has been successfully applied in the solution phase. It has been shown that the non-symmetric form-factor matrix can be efficiently transformed into a symmetric and positive definite matrix to be used in the SCG method. It has been experimentally observed that the SCG algorithm converges faster than commonly used Gauss–Jacobi (GJ) algorithm, which converges in almost double the number of iterations of the SCG algorithm. Efficient parallel SCG and GJ algorithms were proposed and implemented. An almost perfect load balance has been achieved by a new and efficient parallel data redistribution scheme. Our experiments verify that the efficiency of the SCG algorithm with the data redistribution scheme remains over 86% for sufficiently large granularity (i.e., $M/P > 11\,148$). We conclude that the SCG method is a much better alternative to the conventional GJ method for the parallel solution phase.

In this paper, several parallel algorithms for the form-factor computation phase were also presented. It has been illustrated that it is possible to reduce the interprocessor communication by sharing the memory space for rows of the form-factor matrix with global patch data. It has also been observed that the demand-driven approach, in spite of its extra communication overhead, achieves better load balancing and hence better processor utilization than static assignment.

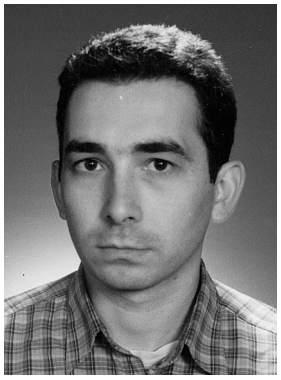
An efficient parallel renumbering scheme has also been proposed to avoid double indexing required in the matrix vector products in the parallel SCG and GJ algorithms when demand-driven assignment is used in the form-factor computation phase.

Acknowledgements. This work is partially supported by the Commission of the European Communities, Directorate General for Industry under contract ITDC 204-82166, and The Scientific and Technical Research Council of Turkey (TÜBİTAK) under grant EEEAG-160.

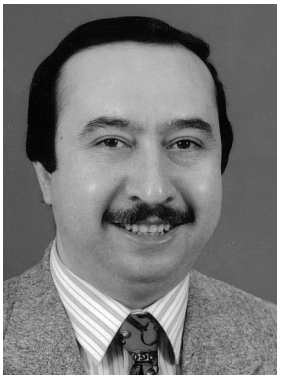
References

1. Aykanat C, Özgüner F, Erçal F, Sadayappan P (1988) Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. IEEE Trans Comput 37:1554–1568
2. Aykanat C, Çapin TK, Özgüç B (1996) A parallel progressive radiosity algorithm based on patch data circulation. J Comput Graph 20:307–324
3. Baum DR, Rushmeier HE, Winget JM (1989) Improving radiosity solutions through the use of analytically determined form-factors. Comput Graph 23:325–334
4. Chalmers AG, Paddon DJ (1989) Implementing a radiosity method using a parallel adaptive system. Proceedings of the 1st International Conference on Applications of Transputers, Liverpool, UK
5. Chalmers AG, Paddon DJ (1990) Parallel radiosity methods. The 4th North American Transputer Users Group Conference, Ithaca, USA, IOS Press, pp 183–193
6. Chalmers AG, Paddon DJ (1991) Parallel processing of progressive refinement radiosity methods. Proceedings of the 2nd Eurographics Workshop on Rendering, Barcelona, Spain
7. Cohen MF, Greenberg DP (1985) The hemi-cube: a radiosity solution for complex environments. Comput Graph (SIGGRAPH'85) 19:31–40
8. Cohen MF, Chen S, Wallace J, Greenberg DP (1988) A progressive refinement approach for fast radiosity image generation. Comput Graph 22:75–84
9. Drucker SM, Schroeder P (1992) Fast radiosity using a data parallel architecture. Proceedings of the 3rd Eurographics Workshop on Rendering, Bristol, UK, pp 247–258
10. Feda M, Purgathofer W (1991) Progressive refinement radiosity on a transputer network. Proceedings of the 2nd Eurographics Workshop on Rendering, Barcelona, Spain
11. Golub GH, van Loan CF (1989) Matrix computations, 2nd edn. The Johns Hopkins University Press, Baltimore, Maryland
12. Goral CM, Torrance KE, Greenberg DP, Battaile B (1984) Modeling the interaction of light between diffuse surfaces. Comput Graph 18:213–222
13. Guitton P, Roman J, Schlick C (1991) Two parallel approaches for a progressive radiosity. Proceedings of the 2nd Eurographics Workshop on Rendering, Barcelona, Spain
14. Hestenes MR, Stiefel E (1952) Methods of conjugate gradients for solving linear systems. Natl Bureau Stand J Res 49:409–436
15. Jájá J, Ryu KW (1992) Load balancing and routing on the hypercube and related networks. J Parallel Distributed Comput 14:431–435
16. Jessel JP, Paulin M, Caubet R (1991) An extended radiosity using parallel ray-traced specular transfers. Proceedings of the 2nd Eurographics Workshop on Rendering, Barcelona, Spain
17. Neumann L (1994) New efficient algorithms with positive definite radiosity matrix. Proceedings of the 5th Eurographics Workshop on Rendering, Darmstadt, pp 219–237
18. Neumann L, Tobler R (1995) New efficient algorithms with positive definite radiosity matrix. In: G. Sakas, P. Shirley, and S. Müller (eds) Photorealistic Rendering Techniques. Springer, pp 227–243
19. Paddon D, Chalmers A, Stuttard D (1993) Multiprocessor models for the radiosity method. Proceedings of the 1st Bilkent Computer Graphics Conference on Advanced Techniques in Animation, Rendering, and Visualization. Ankara, Bilkent Univ., Ankara, pp 85–103

20. Price M, Truman G (1990) Radiosity in parallel. Application of transputers: Proceedings of the 1st International Conference on Applications of Transputers, IOS Press, Amsterdam, pp 40–47
21. Purgathofer W, Zeiller M (1990) Fast radiosity by parallelization. Proceedings of the Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics, Rennes, pp 173–184
22. Ranka S, Sahni S (1990) Hypercube algorithms with applications to image processing and pattern recognition. Bilkent University Lecture Series, Springer, Berlin Heidelberg New York
23. Ryu KW, Jájá J (1990) Efficient algorithms for list ranking and for solving graph problems on the hypercube. IEEE Trans Parallel Distributed Syst 1:83–90
24. Varshney A, Prins JF (1992) An environment-projection approach to radiosity for mesh-connected computers. Proceedings of the 3rd Eurographics Workshop on Rendering, Bristol, UK, pp 271–281
25. Watt A (1989) Fundamentals of three-dimensional computer graphics. Addison Wesley, Wokingham New York Amsterdam Bonn
26. Whitman S (1992) Multiprocessor methods for computer graphics rendering. Jones and Bartlett Boston



TAHSİN M. KURÇ received his B.S. degree in Electrical and Electronics Engineering from the Middle East Technical University, Ankara, Turkey, in 1989 and his M.S. degree in Computer Engineering and Information Science from Bilkent University, Ankara, Turkey, in 1991. He is currently a Ph.D. student at Bilkent University. His research interests include parallel computing and algorithms, parallel computer graphics applications, visualization, and rendering.



CEVDET AYKANAT received his B.S. and M.S. degrees from the Middle East Technical University, Ankara, Turkey, and his Ph.D. degree from Ohio State University, Columbus, all in electrical engineering. He was a Fulbright Scholar during his Ph.D. studies. He worked at the Intel Supercomputer Systems Division Beaverton, as a research associate. Since October 1988 he has been with the Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey, where he is currently an associate professor. His research interests include parallel computer architectures, parallel algorithms, parallel computer graphics applications, neural network algorithms, and fault-tolerant computing.

BÜLENT ÖZGÜÇ joined the Bilkent University Faculty of Engineering, Turkey, in 1986. He is a professor of computer science and the dean of the Faculty of Art, Design and Architecture. He formerly taught at the University of Pennsylvania, USA, Philadelphia, College of Arts, USA, and the Middle East Technical University, Turkey, and worked as a member of the research staff at the Schlumberger Palo Alto Research Center, USA. For the last 15 years, he has been active in the field of computer graphics and animation. He received his B. Arch. and M. Arch. in architecture from the Middle East Technical University in 1972 and 1973. He received his M.S. in architectural technology from Colombia University, USA, and his Ph.D. in a joint program of architecture and computer graphics from the University of Pennsylvania in 1974 and 1978, respectively. He is a member of ACM Siggraph, IEEE Computer Society and UIA.