Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Computer Science
Bratislava, Slovakia

Parallelization of Radiosity Method

Master's Thesis

Pavol Slamka
author

Dr. Tomáš Plachetka
advisor

Bratislava May 2007

Parallelization of Radiosity Method

Master's Thesis

Pavol Slamka

Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Computer Science
Bratislava, Slovakia

Study Programme: 2508800 Informatics

Dr. Tomáš Plachetka

Bratislava, May 2007

I hereby declare that this thesis is my own work, and was written only with the help of the referenced literature and under the careful supervision of my thesis advisors.

Bratislava, May 2007 Pavol Slamka

Acknowledgements

I would like to thank my teacher and consultant *Dr. Tomáš Plachetka* for his time, worthy advice and guidance on this diploma thesis. I would also like to thank *Prof. RNDr. Branislav Rovan PhD.* for his comments on this thesis.

Abstract

In this thesis, we focus on design of a highly efficient parallel algorithm for solving the radiosity method. The presented parallel algorithm is based on the progressive radiosity method. We use asynchronous message passing model in combination with communication-thread in order to overlap communication with computation. This approach allows us to take a different numerical approach to the problem, thus decreasing the communication between processors. All this improves efficiency of the parallel algorithm. The parallelization at a high level of sequential algorithm further decreases the communication and can be viewed as a parallel framework, which is independent of the implementation of lower layers. In this thesis, we design a fully asynchronous algorithm and an algorithm with deferred synchronization for progressive radiosity solution. For comparison, we also designed a simple synchronous algorithm. Finally we propose two load balancing approaches based on work stealing, which are suitable for presented algorithms.

Keywords: Parallelization, Progressive radiosity, Asynchronous model

Contents

1 Introduction

Virtual reality has become a common part of today's life. We meet with new amazing worlds and possibilities and that not just in the movie industry or computer games, but also in art or advertising. The visualization of artificial space is an inextricable part of virtual reality. Creation and visualization of three-dimensional scenes is also used for simulations and has its significance in other sectors of industry, e.g. architecture.

Photorealistic image synthesis is a part of computer graphics, which is examining the possibilities of true visualization of existent or non-existent three-dimensional scenes with the help of a computer. Requirements for the output's "realism" differ according to the field of use. In general, however, the exact solution is never reached and the result is just its approximation. The greater accuracy we want to achieve, the greater are the demands for the computation.

In spite of today's permanent growth of speed and capabilities of computers, there are physical boundaries. Therefore, the growth of computer's power will eventually stop. Moreover, the increase of speed of computers is still relatively slow. Although not all computations can be solved in parallel with a significant efficiency increase, parallel computing is a natural way for further increase of computational power.

Radiosity is one of the methods of photorealistic image synthesis. With increasing accuracy and quality of the output image, the computational demands are increasing incomparably faster. For this reason, radiosity is a good candidate for parallelization. In this work, we will focus on the effective, high-level parallelization of the radiosity method, which exploits an asynchronous model of computation and communication, and therefore also modifies the way of numerical solution.

1.1 Radiosity and Current Research

Radiosity as a method of photorealistic synthesis became very popular. Because of the ability to simulate indirect light and creating soft shadows, rendered images have very realistic looks, in spite of several drawbacks. To remedy some of these drawbacks, radiosity can be combined with other methods such as ray tracing, in a so-called two-pass solution [WCG87] [SP89], to produce even better quality images.

The recent research in the radiosity field is focusing on two goals. The first one is to increase the accuracy and correctness of the resulting image, so that the results have fewer artifacts, look more realistic and are correct within the physical model. The second goal researchers try to achieve is to speed up the computation and optimize the amount of resources needed, since the radiosity computation is, similar to other photorealistic image synthesis methods, very time and memory consuming. Some approaches are successful in improving both these goals. Research involves topics such as input scene representation, adaptive subdivision, probabilistic models for visibility computation, dynamic scenes, realtime radiosity computation, or parallelization.

1.2 Parallelization of Radiosity

There have been many different approaches and strategies to the parallelization of radiosity. Although it is quite easy to develop an arbitrary parallel algorithm for radiosity, it is a difficult task to develop an efficient one.

Former parallelization approaches differ in the choice of architecture, where the topology and communication overhead play an important role, in the definition and distribution of tasks, and in special hardware usage. Moreover, different basic sequential radiosity algorithms were chosen for parallelization. This may have a significant influence on how parallelism can be exploited and on the level it can be exploited at. The basic algorithms differ in the choice of techniques used for local computations, such as visibility computation, in the choice of the method for solving linear equations system and in optimization techniques used.

The main challenge in parallelization is to avoid situations, where there are unoccupied processors waiting for other processors, thus wasting the computational time. Also, we try to use as little communication between processors as possible, because communication is slow[1] and introduces additional cost in comparison with sequential algorithm. In our opinion, many radiosity algorithms still lack a straightforward approach to achieving these goals.

It is a common practice to write parallel applications using one of two standard parallel libraries, MPI or PVM. Applications using these libraries can be easily modified to run at different architectures, what outweighs a small performance loss. Unfortunately, these parallel standards show an inconvenient drawback when used with a certain class of parallel applications, as it was addressed in [Pla06] or [Pla03]. When using PVM or MPI libraries for the class of non-trivial parallel application, active polling (or busy waiting) occurs, causing inefficient computation. In our application, the use of threads is necessary in order to exploit the asynchronous model. There are several thread-safe MPI implementations. However, the active polling problem still occurs, either hidden in the library implementation, or at the application level. The result is a significant loss of efficiency. The solution to this problem, thread parallel library, which is thread-safe and avoids the active polling problem, was proposed in [Pla06]. The use of this library allows us to exploit a full asynchronous communication. The asynchronous communication is very efficient — however, it also brings new implementation challenges in comparison to synchronous communication.

1.3 Outline of This Thesis

In this thesis we present an efficient asynchronous algorithm for the radiosity method, based on modified progressive radiosity, using the asynchronous message passing model, and we avoid the active polling problem by using

[1]Compared to the processor speed

thread-safe, non-polling communication library.

In chapter 2 we present radiosity as a method for realistic image synthesis. We start by describing the problem of global illumination and continue with the definition of radiance and radiance equation. Next, we focus on the derivation of radiosity equation. We also discuss the possibilities to the solution of the radiosity equation where we look more in detail at the progressive radiosity, which is the base for the presented parallel algorithms. In the third part of chapter 2, we consider possible approaches to form factor computation. The objectives, difficulties and different approaches to parallelization of radiosity method are discussed in chapter 3. In chapter 4 we first acquaint the reader with assumptions made to the presented algorithms. We describe a simple synchronous algorithm and a fully asynchronous algorithm followed by an algorithm with deferred synchronization. We also prove the termination and convergency properties for the asynchronous algorithm. In chapter five we discuss the possibilities if improving the presented algorithms using load balancing. We offer two approaches based on work stealing technique.

2 Radiosity Method

2.1 Global Illumination

Given an arbitrary 3D model, which consist of the geometry of individual 3D objects, the material properties for these objects, light sources and the virtual camera, the problem of global illumination can be formulated as computing the image that the virtual camera takes, according to laws of physics. This problem can be divided in two independent phases:

- computation of the light distribution in the scene
- measurement of the light distribution by the virtual camera

The first phase computes the distribution of light according to the laws of physics. Note that the first phase is independent on the virtual camera position. If the light distribution computed in the first phase is stored (as in the case of radiosity), then it is easy to compute pictures as the virtual camera moves and measures light information from different positions. In order to formally define the global illumination problem, we will introduce several definitions:

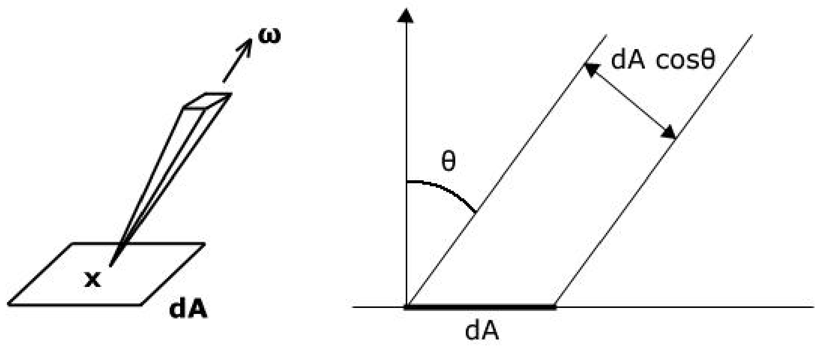
Definition 1. *Radiant power (flux) Φ is the amount of energy which passes through a boundary per unit time (over a given spectrum range).*

$$\Phi = \frac{dQ}{dt} \tag{1}$$

Definition 2. *Radiance (intensity) is the radiant power radiated (or received) at a given point x in a given direction $\overrightarrow{\omega}$, per unit projected surface area $dA^{\perp}$[2], per unit solid angle $d\overrightarrow{\omega}$, over a given spectrum range (Fig. 2(a)).*

$$L(x, \overrightarrow{\omega}) = \frac{d^2\Phi}{dA^{\perp} d\overrightarrow{\omega}} \tag{2}$$

[2]Per unit area perpendicular to the direction of travel



(a) Radiance leaving differential area dA in the direction ω.

(b) Unit projected surface area.

Figure 1: Radiance.

The unit projected surface area can be expressed as:

$$dA^{\perp} = dA \cos\theta \tag{3}$$

where θ is the angle between the local surface normal and the direction $\overrightarrow{\omega}$ (Fig. 2(b)).

The importance of radiance lies in the fact that human visual perception is sensitive to radiance. The same is valid for a camera sensor. The intensity, the color of a pixel is proportional to the radiance of the corresponding object. In vacuum, radiance in particular direction is invariant everywhere along this direction [CWH93]. Therefore the color or contrast of an observed object does not change with changing distance of the observer. Radiance has also the advantage that many radiometric quantities can be expressed as radiance integrals.

Definition 3. *Radiosity is defined as the radiant power radiated*[3] *from a differential surface area around a given point.*

$$B(x) = \frac{d\Phi}{dA} \tag{4}$$

[3]This includes both reflected flux and emitted flux

Radiosity can be expressed as the radiance integral over the hemisphere.

$$B(x) = \int_{\Omega} L(x, d\overrightarrow{\omega}) \cos\theta d\overrightarrow{\omega} \tag{5}$$

Definition 4. *The bidirectional scattering function, BSDF is defined as the ratio of the scattered radiance and the incoming radiance:*

$$BSDF(x, \omega', \omega) = \frac{dL_s(x, \omega)}{dE_i(x, \omega)} = \frac{dL_s(x, \omega)}{dL_i(x, \omega') \cos\theta' d\omega'} \tag{6}$$

where x is the point of incidence, ω is a differential solid angle around the outgoing direction, ω' is a differential solid angle around the incoming direction, θ' is the angle between the surface normal and the incoming direction.

BSDF is a more general function than BRDF[4] because it describes both the reflection and the refraction of light. Given the BSDF function for a surface and the value of incoming radiance for a particular point x from a particular direction ω', we can determine, for any outgoing direction ω, the value of scattered radiance using the *scattering equation*, which describes the *local illumination model*[CWH93]:

$$L_s(x, \omega) = \int_{\Omega} BSDF(x, \omega', \omega) L_i(x, \omega') \cos\theta' d\omega' \tag{7}$$

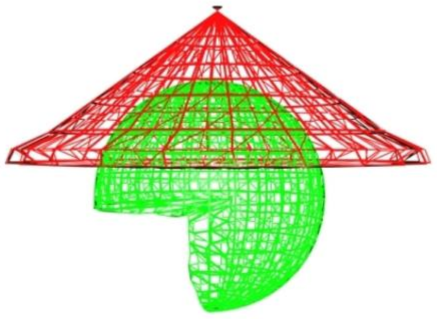
Surfaces are usually modeled using polygonal mesh, Fig. 3(a), or constructive solid geometry, which uses boolean operators on simple objects in order to create complex surfaces as shown in Fig. 3(b). For more elaborate information, you can refer to [ZBSF04] or [Pla03].

As stated in [Pla03], we can formally define an instance of the global illumination.

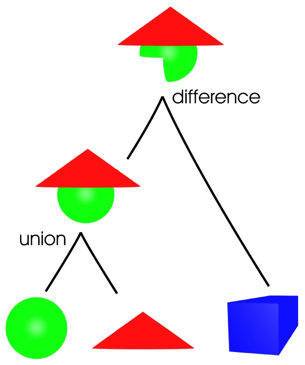
Definition 5. *The instance of the global illumination problem is a tuple*

$$\langle G, BSDF, L_e, C \rangle \tag{8}$$

[4]Bidirectional reflectance distribution function



(a) An example of triangle mesh. The object consists of many connected triangles.



(b) Description of an object in constructive solid geometry. The object is a result of the difference and union operations.

Figure 2: Surface modeling.

where G is the description of surfaces, $BSDF$ is the function describing the surface materials, L_e is the description of light sources and C is the virtual camera description.

In order to solve the global illumination problem, we have to compute at least the radiance values of the scene objects which are visible by the virtual camera. The camera can afterwards record the picture according to the radiance values.

Definition 6. *The Ray-Trace function $RT(x, \omega)$ returns the nearest surface point to x in the direction ω. (If no surface point is found in direction ω, an arbitrary point along the direction ω from the point x is returned.)*

Considering that the total radiance consists of the emitted radiance and the scattered radiance and using (7) and the Ray-Trace function $RT(x, \omega)$, we get the *Radiance equation*:

$$L(x, \omega) = L_e(x, \omega) + L_s(x, \omega) \tag{9}$$

$$L(x,\omega) = L_e(x,\omega) + \int_\Omega BSDF(x,\omega',\omega)L(RT(x,-\omega'),\omega'))\ cos\theta' d\omega' \quad (10)$$

The radiance equation can be solved by using direct methods, or approximation methods. The former use Monte Carlo methods for direct integration of the equation, e.g. the gathering path method. The latter, e.g. the radiosity method, introduce simplifications to the original model in order to simplify the radiance equation.

2.2 Radiosity Equation

Radiosity method historically originates from radiative heat transfer computation [GTGB84]. It is one of more ways to find the solution for the basic illumination equation, however, introducing several simplifying assumptions as follows. The first simplification is, that all of the scene surfaces are perfectly diffuse reflectors, also called Lambertian surfaces, reflecting light equally in all directions. By all directions we only mean directions in the half space of reflection. We can characterize a perfect diffuse surface by BSDF as follows:

$$BSDF(x,\omega',\omega) = \begin{cases} \frac{\rho(x)}{\pi} & \text{if } \omega \text{ lies in the half-space of reflection;} \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

where $\rho(x) \in \langle 0,1 \rangle$ is the reflection coefficient.

This simplification, of course, brings errors to the rendered images, because as we know, none of real objects is perfectly diffuse. Moreover, there are objects, which we cannot describe using diffuse reflection, not even approximately, e.g. mirror or transparent objects. We can neither simulate refraction of light at the boundaries of different optical environments, as we can in ray tracing.

The second simplification would be, that the 3D scene consists of one type of objects only. These objects, so called *patches*, are planar polygons, which represent parts of the surface of an object in the scene. We consider every patch being uniform in all its surface points, in the sense of the BSDF and the radiance values. Patches also represent light sources. All light sources are area light sources, and are perfect diffuse emitters. Thanks to these simplifications, the radiance equation can be simplified into a more feasible form, and so can be the radiosity equation. First, we rewrite the equation (10) using (11):

$$L(x,\omega) = L_e(x,\omega) + \frac{\rho(x)}{\pi} \int_\Omega L(RT(x,-\omega'),\omega'))\ cos\theta' d\omega' \tag{12}$$

Definition 7. *We define excitance as:*

$$E(x) = \int_\Omega L_e(x,\omega) \cos\theta d\omega \tag{13}$$

where $L_e(x,\omega)$ is the function describing only radiance outgoing from light sources in the scene[5]. Since all surfaces are Lambertian surfaces, $L_e(x,\omega)$ is not dependent on ω and we can simplify the previous equation using spherical coordinates:

$$\begin{aligned} E(x) &= \int_\Omega L_e(x,\omega) \cos\theta d\omega \\ &= L_e(x,\omega) \int_\Omega \cos\theta d\omega \\ &= L_e(x,\omega) \int_0^\pi \int_0^{2\pi} \cos\theta \sin\theta d\theta d\phi \\ &= \pi L_e(x,\omega). \end{aligned} \tag{14}$$

We can also simplify the definition of the radiosity quantity (3):

[5]Given a point x that does not lie inside any light sources, $E(x) = 0$

$$\begin{aligned} B(x) &= \int_\Omega L(x,\omega)\cos\theta d\omega \\ &= L(x,\omega)\int_\Omega \cos\theta d\omega \\ &= \pi L(x,\omega). \end{aligned} \tag{15}$$

After multiplying (12) by π we obtain:

$$B(x) = E(x) + \rho(x)\int_\Omega L(RT(x,-\omega'),\omega')\cos\theta' d\omega' \tag{16}$$

The Ray-Trace function returns the closest surface point y in the direction $-\omega'$ if there is one (if no surface point is found in direction $-\omega$, an arbitrary point y_{arb} along the direction $-\omega$ from the point x is returned, thus $L(y_{arb},\omega) = 0$):

$$y = RT(x,-\omega'). \tag{17}$$

Since we know that y lies on a surface that is a perfect diffuse reflector and emitter, it holds that

$$L(y,\omega') = L(RT(x,-\omega'),\omega') = \frac{B(y)}{\pi} \tag{18}$$

Now we can see from (18) that the radiance incoming from a point of surface depends only on the radiosity of this point. Therefore, the equation (16) can be further simplified if we substitute for $d\omega'$[6]:

$$B(x) = E(x) + \frac{\rho(x)}{\pi}\int_S B(y)\frac{\cos\theta\cos\theta'}{r(x,y)^2}V(x,y)dA \tag{19}$$

where θ is the angle between the surface normal and the direction $y \rightarrow x$ at the point y, $r(x,y)$ is the distance function, dA is the differential area around point y. Because we integrate through all surface points S in the scene, we

[6] $d\omega' = dA\cos\theta/r^2$

must cut off those points which are not visible from x and therefore don't contribute to the incoming radiance. This is the reason why the visibility term $V(x, y)$ is introduced.

$$V(x,y) = \begin{cases} x & \text{if } x \text{ and } y \text{ are mutually visible;} \\ 0 & \text{otherwise.} \end{cases} \tag{20}$$

The surfaces of the scene S are represented by patches P_i. That allows us to split the integration domain and integrate over all patches, since $S = \bigcup_{i=1}^{n} P_i$:

$$B(x) = E(x) + \frac{\rho(x)}{\pi} \sum_{i=1}^{n} \int_{y \in P_i} B(y) \frac{\cos\theta \cos\theta'}{r(x,y)^2} V(x,y) dA_i \tag{21}$$

We suppose that all patches are uniform, hence all the points of a particular patch have the same radiosity and material properties. Since the incoming radiance, and therefore also the radiosity value, varies over the points of a particular patch, we introduce the *patch radiosity* as the area-weighted average of point radiosities, in order to satisfy the uniformity condition.

$$B_i = \frac{1}{A_i} \int_{x \in P_i} B(x) dx \tag{22}$$

Similarly, we obtain uniformity of excitance over a patch.

$$E_i = \frac{1}{A_i} \int_{x \in P_i} E(x) dx \tag{23}$$

After incorporating patch uniformity to the equation (21), we obtain

$$B_i = E_i + \rho_i \sum_{j=1}^{n} Bj \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta \cos\theta'}{\pi r(x,y)^2} V(x,y) dA_j dA_i \tag{24}$$

where $\rho_i = \rho(x)$ for all x such that $x \in P_i$.

Let us denote

$$F_{ij} = \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta \cos\theta'}{\pi r(x,y)^2} V(x,y) dA_j dA_i \tag{25}$$

where the F_{ij} terms are called form factors. Then the radiosity equation can be written as linear equation system

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} Bj \tag{26}$$

with a corresponding matrix form

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & \vdots \\ \vdots & \dots & \ddots & \vdots \\ -\rho_n F_{n1} & \dots & \dots & 1-\rho_n F_{nn} \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} \tag{27}$$

The solution of the global illumination problem with the radiosity method consists of two steps. In the first (view-independent) step, we solve the radiosity equation in order to find radiosity values for all patches in the scene. In the second step, we look at the scene with the camera from a certain position and we produce a corresponding image. The first step is completely independent of the camera position, hence we might repeat the second step placing the camera at different positions, using the radiosity values computed in the first step. In this fashion, animated sequences can be created efficiently. Note that we change only the position of the camera in the scene, while all objects of the scene (patches) remain at the same position. Once we move an object and change the geometry of the scene, the first step needs to be recomputed again, although techniques exists which try to avoid this [Sch00].

In order to find the solution of the radiosity equation, we need to compute the form factors F_{ij}, which are only geometry dependent, and can be therefore pre-computed, if needed. Next, we solve the equation, using an appropriate method for solving linear equation systems. However, the computation of form factors is very time-consuming. With the increasing number of patches in the scene, the size of the radiosity equation matrix grows with square of the number of patches in the scene. Hence, for large scenes, those methods for solving linear equations, which need to have the whole matrix computed, e.g. Gauss elimination, Jacobi or Gauss-Seidel, are not very practical. Instead, other methods are used, such as Southwell relaxation, which is related to the progressive radiosity method.

2.3 Progressive Radiosity

Definition 8. *We say that a given matrix M is strictly row diagonally dominant, if $\forall i : |M_{ii}| > \sum_{j \neq i} |M_{ij}|$.*

Definition 9. *We say that a given matrix M is strictly column diagonally dominant, if $\forall j : |M_{jj}| > \sum_{i \neq j} |M_{ij}|$.*

For the numerical solution of the radiosity equation

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} Bj \tag{28}$$

several relaxation methods can be used[7]. We refer to the solution of equation (28), using Gauss-Seidel method, as to *gathering radiosity*, because the radiosity B_i of the patch P_i is defined as a sum of other patches' radiosities, in a sense, it is gathering the incoming light. To find a better estimation for a single patch's P_i radiosity B_i, we need to compute the i^{th} row of the matrix from (27). In order to determine this one row, we need to compute $O(n)$ form factors (where n is the number of patches). Furthermore, to estimate the radiosity of all patches using the Gauss-Seidel method, at least the first complete iteration cycle is necessary [CCWG88]. That means, we need to compute the whole matrix — $O(n^2)$ form factors. This is intractable even for

[7]Matrix from equation (27) is row diagonally dominant, because $F_{ii} = 0$, $\sum_{j=1}^{n} F_{ij} \leq 1$ and $0 \leq \rho_i < 1$

moderately large scenes. Moreover, this matrix may be too large to fit into memory. A method that avoids these drawbacks, progressive radiosity (also called *shooting radiosity*), was proposed [CCWG88] as an approach to the solution of radiosity equation following the progressive refinement principle [BFGS86], and it is equivalent to the combination of Southwell relaxation and Jacobi iteration [GCS93].

Progressive radiosity works in iterations, which are repeated until the desired convergence criterion is met. In every iteration, we first select the patch with the greatest unshot energy.[8] Afterwards, this patch — *shooter* "shoots" its energy on other patches, *receivers*. The energy is distributed among the receivers according to the corresponding form factors. For every patch, we store the total received radiosity, B_i, and the radiosity to be shot, R_i. When increasing the value of B_i, we increase the total "brightness" of patch P_i. When increasing residual R_i, we increase the unshot energy, which should be reflected from patch P_i and hit other patches. After patch P_i shoots its energy that corresponds to $R_i A_i$ (where A_i is the area of patch P_i), its residual R_i is set to zero and the iteration ends. After the algorithm has finished, the resultant radiosities are found in B_i. Note, that we only compute the form factor F_{ij} when needed, and we do not store it. In comparison with full matrix methods, we avoid storing $O(n^2)$ form factors. On the other hand, we may sometimes have to compute the same form factor again.

```
/*INPUT*/
  var array E[i]; /*light sources radiosities*/
  var array r[i]; /*patch reflectances*/
  var array P[i]; /*patch geometry description*/

/*VARIABLES*/
  var array B[i]; /*total radiosities*/
  var array R[i]; /*residual radiosities*/
```

[8] In the beginning of algorithm, this will be the strongest light source

```
BEGIN

for all i:
{
   B[i] := r[i] * E[i];
   R[i] := r[i] * E[i];
}

while not converged
{
  select i with greatest R[i]
  for all j
  {
      F = Compute_Form_Factor(P[i], P[j]);
      increase = r[j] * F * R[i]A[i] / A[j];
      E[j] := E[j] + increase;
      R[j] := R[j] + increase;
  }
  R[i] := 0;
}

END
```

In the next few lines, we will explain the principle of the Southwell relaxation. We will also prove that the Southwell relaxation is valid for equation systems with column diagonally dominant matrix.

Consider a linear equation system

$$Ax = b, \tag{29}$$

where the matrix M is $n \times n$ and is column diagonally dominant. In Southwell relaxation, we construct a sequence of approximate solutions for vector x, which converges to the system's solution. We denote $x^{(k)}$ the approximate solution after the k^{th} step. We define the k^{th} residual as

$$res^{(k)} = b - Ax^{(k)}. \tag{30}$$

In every step, we update the approximate solution

$$x^{k+1} = x^k + \Delta x. \tag{31}$$

At the beginning, we set $x^{(0)} = 0$, therefore $res^{(0)} = b$. Let us consider that after the k^{th} step we are given vector $res^{(k)}$. Then we can compute $res^{(k+1)}$:

$$res^{(k+1)} = b - Ax^{(k+1)} = b - A(x^k + \Delta x) = res^{(k)} - A\Delta x. \tag{32}$$

Following the Southwell relaxation, we select the component from $res^{(k)}$ having the greatest residual. Our goal is to relax this i^{th} component's residual $res_i^{(k)}$ so that $res_i^{(k+1)} = 0$. From (32), we get

$$\begin{aligned} res_i^{(k+1)} &= res_i^{(k)} - \sum_{j=1}^{n} A_{ij}\Delta x && (33)\\ 0 &= res_i^{(k)} - \sum_{j=1}^{n} A_{ij}\Delta x && (34)\\ res_i^{(k)} &= \sum_{j=1}^{n} A_{ij}\Delta x. && (35) \end{aligned}$$

The following vector is solution to the equation above

$$\Delta x_j = \begin{cases} 0 & \text{if } i \neq j; \\ res_i^{(k)}/A_{ii} & \text{if } i = j. \end{cases} \tag{36}$$

After updating the approximate solution vector by Δx as defined above, we update residua as follows

$$res_j^{(k+1)} = res_j^{(k)} - \frac{A_{ji}}{A_{ii}} res_i^{(k)}. \tag{37}$$

The proof of convergence follows:

Proof. In this proof, we will use the norm

$$||res^{(k)}|| = \sum_i |res_i^k|. \tag{38}$$

Note that if the convergence holds for the norm as defined above, it also holds for the max-norm, which is defined as

$$||res^{(k)}|| = MAX|res_i^k|. \tag{39}$$

We know from (37)that

$$res_j^{(k+1)} = res_j^{(k)} + (\frac{-A_{ji}}{A_{ii}} res_i^{(k)}). \tag{40}$$

Since $|a+b| \leq |a| + |b|$, we obtain

$$|res_j^{(k+1)}| \leq |res_j^{(k)}| + \frac{|A_{ji}|}{|A_{ii}|} |res_i^{(k)}|, \tag{41}$$

we sum up for all $j \neq i$

$$\sum_{j \neq i} |res_j^{(k+1)}| \leq \sum_{j \neq i} |res_j^{(k)}| + \sum_{j \neq i} \frac{|A_{ji}|}{|A_{ii}|} |res_i^{(k)}|, \tag{42}$$

$$\sum_j |res_j^{(k+1)}| - |res_i^{(k+1)}| \leq \sum_j |res_j^{(k)}| - |res_i^{(k)}| + \frac{|res_i^{(k)}|}{|A_{ii}|} \sum_{j \neq i} |A_{ji}|, \tag{43}$$

$$||res^{(k+1)}|| \leq ||res^{(k)}|| - |res_i^{(k)}| + \frac{|res_i^{(k)}|}{|A_{ii}|} \sum_{j \neq i} |A_{ji}|. \tag{44}$$

Let us denote

$$\lambda_i = \frac{\sum_{j \neq i} |A_{ji}|}{|A_{ii}|} \tag{45}$$

From the strict column diagonal dominance of A we derive that

$$0 \leq \lambda_i < 1. \tag{46}$$

The previous inequality is valid for any particular i. If we denote λ the maximum among all λ_i, then the following holds

$$||res^{(k+1)}|| \leq ||res^{(k)}|| - |res_i^{(k)}| + |res_i^{(k)}|\lambda, \tag{47}$$

where $0 \leq \lambda < 1$.

$$||res^{(k+1)}|| \leq ||res^{(k)}|| + |res_i^{(k)}|(\lambda - 1). \tag{48}$$

Since $res_i^{(k)}$ is the greatest residual, it holds that

$$|res_i^{(k)}| \geq \frac{||res^{(k)}||}{n}. \tag{49}$$

Therefore, since $(\lambda - 1)$ is negative, we can exchange the term $|res_i^{(k)}|$ and we obtain

$$||res^{(k+1)}|| \leq ||res^{(k)}||(1 + \frac{\lambda - 1}{n}). \tag{50}$$

$1 + \frac{\lambda-1}{n} < 1$, hence

$$||res^{(k+1)}|| \leq ||res^{(k)}||q \tag{51}$$

where q $<$ 1.

$$||res^{(k+1)}|| \leq ||res^{(0)}||q^k \tag{52}$$

Since q^k converges to 0, the residua converge to 0, the vector of unknowns converges to the solution. $\square$

We can see that in the progressive radiosity algorithm, radiosities and residua of all patches are updated in one shooting step. In the Southwell relaxation, in one relaxation step we update all residua, but only one component of the unknown vector.[9] The permanent updating of radiosity values of all patches in every iteration step is equivalent to one use of Jacobi sweep at the end of Southwell relaxation. More specifically, once we are done with Southwell relaxation, we still have the computed values of the residua. The addition of these residua to the solution vector is in fact one Jacobi sweep. Every Southwell relaxation step adds the residua to the relaxed component, $x_i^{(k+1)} = x_i^{(k)} + res_i^{(k)}/A_{ii}$. If we were running Southwell long enough, all residua would be eventually added to corresponding components, but updated in every step. We do the same with the Jacobi sweep, just using the old data, because the residua are not updated.[10] The use of Jacobi is valid for systems with strictly diagonal matrix, as noted in [GCS93]. It follows, that the progressive radiosity converges to the solution of equation (55).

From (25) we can derive the reciprocity relation for form factors

$$F_{ij}A_i = F_{ji}A_j \tag{53}$$

where A_i and A_j are areas of patches P_i and P_j, respectively. After multiplying (28) by term A_i, the reciprocity relation allows us to rewrite it as

$$B_iA_i = E_iA_i + \rho_i \sum_{j=1}^{n} BjA_jF_{ji} \tag{54}$$

[9]Which corresponds to the radiosity value of one patch

[10]In case of radiosity, the old residua are smaller than the new ones would be, if we continued with Southwell relaxation long enough. Therefore, by using not-updated residua, we move closer to the solution, just more slowly than if we had the updated residua.

with the matrix form

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{21} & \dots & -\rho_1 F_{n1} \\ -\rho_2 F_{12} & 1-\rho_2 F_{22} & \dots & \vdots \\ \vdots & \dots & \ddots & \vdots \\ -\rho_n F_{1n} & \dots & \dots & 1-\rho_n F_{nn} \end{pmatrix} \cdot \begin{pmatrix} B_1 A_1 \\ B_2 A_2 \\ \vdots \\ B_n A_n \end{pmatrix} = \begin{pmatrix} E_1 A_1 \\ E_2 A_2 \\ \vdots \\ E_n A_n \end{pmatrix} \tag{55}$$

.

Lemma 2.1. *The matrix M from (55) is column diagonally dominant.*

Proof. Since $\forall i F_{ii} = 0$, it follows that $\forall i M_{ii} = 1$. From the physical constraints we have that $\forall i \sum_{j \neq i} F_{ij} <= 1$ (for closed scenes, the equality is true, i.e. $\sum_{j \neq i} F_{ij} = 1$). Also, $0 \leq \rho_i < 1$. Therefore $\forall i \sum_{j \neq i} |M_{ij}| < 1$. Because $\forall i M_{ii} = 1$, the lemma holds. □

Gauss Seidel and Southwell relaxations are equivalent to gathering and shooting radiosity, respectively. While the Southwell relaxation has the advantage of the greatest residual choice and exhibits better estimates in the beginning[11], in the long run,[12] the greedy choice is unnecessary and the Gauss-Seidel method will work at least as well as the Southwell relaxation. Progressive radiosity as a combination of Southwell and Jacobi allows us to get an early approximation for all patches that can be computed using $O(n)$ form factor computations only. Also, it allows us to compute the form factors when needed, avoiding their explicit storage. Finally, not all $O(n^2)$ form factors have to be determined. Therefore, we have chosen the progressive radiosity algorithm as the base for our parallelization.

2.4 Form Factor Computation

The most expensive part in the radiosity computation is the visibility computation. To compute the fraction of shooter's[13] energy that should be trans-

[11] In the radiosity sense, it shoots the light sources in the beginning.
[12] When we run $O(n)$ iterations.
[13] We use the shooter / receiver analogy from the progressive radiosity.

mitted to another particular patch, we need to compute form factor between these two patches. Form factor is designation of the radiosity equation's term

$$F_{ij} = \frac{1}{A_i} \int_{x \in P_i} \int_{y \in P_j} \frac{\cos\theta \cos\theta'}{\pi r(x,y)^2} V(x,y) dA_j dA_i \tag{56}$$

The value of the form factor between shooter and receiver is dependent on the following factors:

- the sizes of the patches, respectively (the larger the receiver, the more energy it will receive, the larger the shooter, the more energy it will shoot)
- the distance between the two patches (the size of the form factor term decreases with the square of distance of the patches)
- the angular position of the patches (if the viewed patch is rotated, it is viewed under smaller angle, a patch not facing the other patch cannot "see" that patch at all)
- the visibility conditions between the patches (there may be obstacles in the scene — other patches, which prevent the shooter from fully "seeing" the receiver, or from "seeing" the receiver at all)

2.4.1 Analytical Methods

The exact value of a form factor can be computed by analytical methods, without errors.[14] However, the use of analytical methods is available only for the simplest configurations, where there are no obstacles between the patches in the scene [How82]. Besides, the patches must be of the simplest shapes. In real scenes, these conditions are met only occasionally, therefore other methods are used to approximate the form factor values.

[14] Consider that with the use of computer, there may be errors because of floating point arithmetic

2.4.2 Projection Methods

Another way to compute form factors is using projection methods. Here, we use the geometrical analogy to form factor, called Nusselt's analogy: For a finite area, the form factor is equivalent to the fraction of the circle (which is the base of the hemisphere) covered by projecting the area onto the hemisphere and then orthographically down onto the circle (Fig. 3). A well known projection method is the hemicube algorithm [CG85]. It constructs a meshed hemicube above the particular patch P_i and pre-computes the delta form factor for every polygon from the mesh. To determine the form factor F_{ij}, we project the patch P_j onto the hemicube and sum up the delta form factors for every mesh polygon the projection covers. However, this doesn't consider possible occlusions. Therefore, we first project all patches onto the hemicube. In case that more different patches project onto the same polygon, we make the proximity test and choose only the nearest one to "occupy" the polygon. Afterwards, we compute the form factors F_{ij} for every P_j, summing up the corresponding delta form factors "occupied" by patch P_j. The drawback of hemicube method is, that it is hard to determine the proper density of hemicube meshing, or resolution, to maintain necessary accuracy. Also, it approximates the form factors for patch P_i as form factors for it's center point, what introduces large inaccuracies if the patch P_i is relatively large.

2.4.3 Ray-Casting Methods

Ray-casting methods provide form factor approximation using stochastic and numeric approach, known as Monte Carlo integration. Monte Carlo integration estimates the value of an integral by randomly generating sample points and computing the values of the given integrated function in the sample points. The resulting integral is then the average of values computed in the sample points.

Ray casting methods are based on random shooting of rays from a particular patch to determine visibility and form factor estimation. Here we

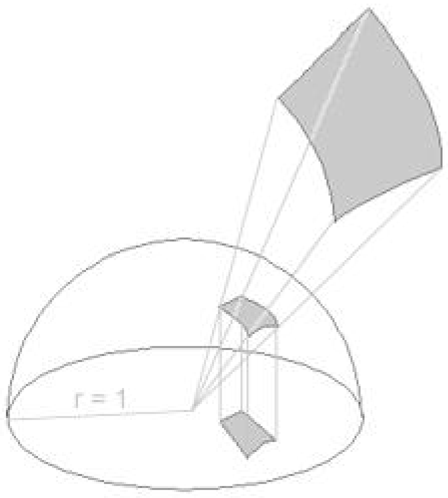


Figure 3: Nusselt's analogy. The patch is first projected onto the unit hemisphere, and afterwards onto the base plane. Form factor is equivalent to the fraction of the covered part of unit circle.

have two basic options. The first one would be, as suggested in [WEH89], to generate sample points at the patch P_i and the patch P_j and connect all the points of patch P_i with all points from the patch P_j, creating "rays". After this, we do intersection tests for all rays to determine, if they collide with an obstacle somewhere between corresponding points of patch P_i and P_j. If there is no intersection, the energy is allowed to flow in the direction of the particular ray and thus the ray contributes to the actual form factor with a small fraction, the delta form factor. If a ray does intersect an obstacle, the contribution is zero. When we sum up the delta form factors' contributions from all rays, we get the form factor between patch P_i and P_j.

We can also follow another idea. In [Mal88], the direction sampling method is proposed. This method generates sample rays leaving from a particular patch P_i in random directions. When such a ray hits a patch P_j in the scene, a delta form factor is added to the F_{ij} form factor. In this fashion, we compute the form factors between P_i and all other patches. Note that there are other approaches, which use the direction sampling and Monte

Carlo integration to directly solve the the integral equation [Kel96], called *random walks* methods.

More detailed description of Monte Carlo methods used in radiosity computation, including estimators and techniques of variance reduction can be found in [Bek99] and [Pie93].

Ray-casting methods can make use of existing techniques of spatial subdivision to speed up the computational times. These techniques can be found in ray tracing algorithms [Gla89]. Well known techniques are the bounding volumes, the bounding volume hierarchies or octrees.

3 Overview of Parallelization Algorithms

3.1 Parallelization Issues

The goal of parallelism is to increase the computational resources available, so that also very demanding computations can be solved at all, or can be solved in a considerably short time.

How do we design a parallel algorithm? Let us say we have a sequential algorithm. Now we could identify tasks processed in the sequential algorithm, divide them into sets among the processors and solve the sets of tasks individually. Unfortunately, some of the tasks may depend on each other. In sequential algorithms, we proceed step by step and those tasks, which other depend on, are always computed first. We construct algorithms naturally in this way — we do it sequentially. However, in parallel algorithms, we do not want the long step-by-step processing, we might want more tasks to be processed at once. Hence, we try to find independent sets of tasks, which can be executed in parallel. Results of these sets of tasks can be the input for other independent sets of tasks that will be processed in parallel just afterwards. In the end, we combine the results of individual processors into the solution, if needed.

The amount of independent tasks may not be enough to feed all of the processors in one parallel step. In that case, we can try to reformulate the problem and use different techniques than in the sequential algorithm that are more suitable for parallelization. For example, in some cases, preprocessing will increase the total amount of computation needed, but will improve the parallelization possibilities, resulting in a better efficiency of the parallel algorithm. The design of efficient parallel algorithms can be very complicated and not very straightforward in comparison with the sequential algorithm. In fact, a good parallel algorithm may be completely different from a sequential one for the same problem. Moreover, some problems are hardly suitable for parallelization at all.

In case of the radiosity method, relatively many tasks are independent and can be identified quite easily. It is questionable, which parts of a computation should be chosen to represent tasks. Let us explain this on a somewhat artificial example.

Imagine a high school where the boys are trying to charm the girls. At this high school, there are only a few boys and lots of girls.[15] Every boy would like to hang out with one of the girls, but the girls are shy. After a talk with a particular girl, every boy can tell his chances. Now every boy talks with all the girls and decides for the one he thinks he has best chances with. Every talk consists of over a hundred topics that are always the same.

Now our task is to simulate all this using N processors and find out for every boy, which girl he is going to ask to go out with him.[16] First, we try to identify the independent tasks, where we have several options. One elementary task could be:

1. one particular boy talks to all girls about all topics
2. one particular boy talks to one particular girl about all topics
3. one particular boy talks to one particular girl about one particular topic

In the first case, we would divide the boys into groups and assign every processor one group.[17] One particular processor will for every boy from his group go over all the girls and talk about all topics, computing the "how-much-she-likes-me" value, and choosing the girl, who likes him the most. In the second case, we will start with the first boy, we will divide the girls among processors and we do the talks with the first boy in parallel. Then the processors will have to agree on the best choice for the first boy.[18] After

[15]There is another high school in town, where almost boys from town go to, because there are many interesting computer science classes.

[16]Since we only simulate the tasks, we make an assumption that both girls and boys can do more talks at once

[17]The number of boys in a group would be approximately the total number of boys divided by N.

[18]Note that this is additional work comparing with the first case, where one processor can tell right away.

that all processors move on to the next boy. In the last case, the processors go over the boys and over the girls and then divide the topics between a particular boy and girl which they simulate. Then, they compute together the "how-much-she-likes-me" value and move on to the next girl. After they are done with all girls for a particular boy, they agree on the most favorable.

At this example, we can see that the tasks can be identified at different levels of the computation. The choice of some levels brings additional costs, because of "common agreements" or *synchronization*, where different processors have to cooperate. This is the case in both the second and the third option of our example.

If we consider, that all the students have different communication skills, then every talk would take a different amount of time. That introduces imbalance to the parallel algorithm, since the tasks themselves are of different difficulty. Consider the second case of our example, where particular processor simulates the talk of a particular boy with a very talkative girl. It will get its answers sooner than other processors, and will have to wait for them in order to agree on (compute) the "how-much-she-likes-me" value. Note, that in the third case, tasks are of equal difficulty, since processors work on topics for a particular boy and girl, where the talking skills are the same.

If the school has very many girls (the problem's data are big), every boy knows only some girls (and their phone numbers). In order to talk with all girls, every boy needs to get all girls' phone numbers. So every boy has to ask the other boys for phone numbers. That means that a processor associated with a particular boy (in the first case of our example), will have to ask another processor, associated with another boy, for the data needed.

3.2 Radiosity Related Problems

From our example, we can extract several issues — task identification, imbalance problems and data locality problems. In radiosity, we have several options where to set up the parallelization. This usually corresponds to the loops in the sequential radiosity algorithm. As we have shown in our example, the parallelization at different levels requires different amounts of overhead, such as synchronization.

The techniques of load balancing are often dependent on the chosen level of parallelization. One approach is the *master — worker* framework, where the master divides the work load among the workers. When the number of processors increases, the master often becomes a bottleneck, since it must process many demands from the workers. Another approach is *work stealing*. In work stealing, the work distribution is not controlled in a centralized fashion. Basically, if there is no work for some processor, it will "steal" the work from another processors, which are loaded with work.

If an algorithm has a bad data locality property — that means every processor has the data it needs — usually a lot of communication is required, decreasing drastically the efficiency of the algorithm.

Finally, in order to achieve best results we often also take the architecture choice into consideration.

3.3 Classification of Parallel Architectures

According to [Cap93], the parallel architectures can be classified according to following criteria:

- number of instructions and data streams supported,
- memory organization,
- coupling,
- granularity.

The classification according to the **number of instruction and data streams** is also known as Flynn's taxonomy [Fly66]:

- **SISD** — Single Instruction Single Data (this is the standard sequential computer),

- **MISD** — Multiple Instruction Single Data,
- **SIMD** — Single Instruction Multiple Data (every processor executes the same instruction, but on its own data),
- **MIMD** — Multiple Instruction Multiple Data (each processor executes different programs with different data)

The present parallel computers are almost exclusively **MIMD** computers. Flynn's classification originates from 1966 and is rather obsolete.

We consider two different classes according to **memory organization**:

- **Shared memory** — all processors have access to global address space. They also may have local memory. Communication between processors is achieved using shared variables.
- **Distributed memory** — all sprocessors have their local memory only — the memory resources are distributed among the processors. Communication between processors is achieved through message passing.

For the coupling criterion, Capin [Cap93] distinguishes two classes:

- **Synchronous architectures** — processors perform their tasks in a lock-step, or highly synchronized manner.
- **Asynchronous architectures** — processors are not synchronized in any fashion, they can process tasks independently at different speeds, sometimes barrier synchronization may be used.

Granularity designates the ratio between the computation and communication time:

- **Coarse-grain architectures** — computation to communication ratio is very high, few powerful processors are used.
- **Medium-grain architectures** — computation to communication ratio of 100 or more.

- **Fine-grain architectures** — computation and communication ratio is almost unity, very large numbers of simple processors are used.

3.4 Previous Work

From the numerical point of view, we are mainly interested in parallel algorithms of progressive radiosity. The radiosity solutions using full matrix methods are suitable only for very accurate computations, where the computation of full matrix is necessary. In all other cases, progressive radiosity exhibits better results. The parallel solutions of full matrix method were presented in [PT89] or [PZ90].

Funkhouser [Fun96] presented an alternative approach using group iterative method. Jacobi group iteration was used. Since the Jacobi method does not use the actual radiosity estimates from other groups,[19] it is suitable for a coarse-grained parallel implementation. In this algorithm processors are assigned group of patches. In every iteration a processor solves its own group to convergence, but with “old data”, since no updates are received from the other groups ,until the end of iteration. The master-slave approach is used, where the master processor assigns groups to the slave processors on demand, until the iteration is finished.

3.4.1 Parallelization of Progressive Radiosity

Most approaches focus on the progressive radiosity method, since it exhibits good results already after linear number of form factor computations. Still there are many different approaches to parallelization of progressive radiosity.

One difference is in the form factor computation. Most of the recent algorithms use the ray-casting technique for the form factor computation [SW], [YIY97], [APRP96] instead of hemicube method [RGG90], [Cap93] which has aliasing effects. However, very good results using the hemicube

[19]The radiosity estimates from the previous iteration are used, even if they are old.

form factor computation were achieved with use of hardware acceleration [BW90]. Some approaches replace the hemicube by a single plane [RGG90].

Considering the Flynn's taxonomy, MIMD computers are used almost exlusively. Parallelization of progressive radiosity on a SIMD computer was proposed in [DS92].

Another important aspect in radiosity computation is the data distribution. When data are replicated among all processors, communication is necessary in order to update the data. In case the data are partitioned and distributed among processors, communication is necessary in case the data needed are not available in local memory. A good technique is to distinguish between the geometry data and the data considering light distribution. Then we can replicate the scene geometry among processors, since geometry is read only, and only distribute the data representing radiosity estimates. We will follow this idea in our algorithms.

Some algorithms strictly follow the sequential progressive radiosity algorithm. For this, however, the patch with greatest residual must be determined and announced to all processors. Therefore, synchronization is necessary. Some algorithms process more than single patch at time. Either all processors select several patches at once together (again synchronization is necessary), or the patches are selected individually in asynchronous fashion. The drawback of parallel processing of several patches is, that not necessarily the best patches are chosen. Still, the asynchronous approach avoids the idle times in synchronization and is in our opinion the better choice.

In [RGG90], only geometry data are duplicated. Since the master-slave approach is used, only the master processor stores the radiosity estimates. The slave processors are on demand assigned a shooter. Every processor carries out the transfer of shooter's energy on all other patches. Several shooters are being shot at the same time. Results are communicated back to the master processor. This technique is called the demand-driven approach. It has the advantage that the computational load is distributed on demand, therefore no loadbalancing is needed. However, the master processor is known to be the bottleneck in case the number of processor increases. The master-slave approach is also used in [FP91], [Šin95], [RGG90] and [SW].

In case the geometry data are distributed, cooperation is often needed to perform the form factor computations. Such approaches usually divide the scene according to some spacial subdivision, and assign every processor one part of the scene — so called *voxel*. Several shooting patches are selected in parallel. Every processor selects the shooter with greatest energy available from its local database. Afterwards it transfers the shooters energy on all other patches. If the patches are not in its local database, it communicates the energy to neighbouring processors.[20] Such approach was proposed in [GRS95]. This approach distinguishes between geometry-data and light-data ownership. In case the non-local light-data were changed, message is sent to owner process. Energy transfer is performed by ray-casting and sending rays to corresponding processes. Communication is improved by sending groups of rays rather then single rays. Multi-thread mechanism is used.

Arnaldi et. al. [APRP96] proposed parallel algorithm using virtual interfaces technique based on virtual walls. Visibility masks are used in order to speed up form factor computations. The algorithm works in asynchronous manner. However, it distributes the geometry among processors, thus increasing the amount of communication. Moreover, the distribution of responsibilities for radiosity estimates is based on spatial subdivision and can lead to severe imbalances [Sch00].

In [BP94] an algorithm using shared virtual memory was proposed, therefore the processors have virtual access to all memory. More patches are processed in parallel. Every processor selects a patch and computes form factors to all other patches. Afterwards the energy distribution is stored in the shared virtual memory. Processors do not synchronize, however concurrent writing problems arise. In case an analogy of our approach for distributed memory was used, memory we be divided into local virtual memories. In-

[20]Neighbours are determined according to the space subdivision.

stead of radiosity updates, only shooters would be "sent" through shared channels. In this fashion, the number of collisions would be decreased.

The duplication of geometry can also be found in [YIY97]. The data corresponding to radiosity estimates are distributed among processors. Mainly the patch distribution in static loadbalancing is discussed, in order to minimize imbalances of computation. The algorithm however follows the synchronized patch selection technique. The synchronous approach is also followed in [Cap93] and [SW].

Good asynchronous algorithm was presented in [Sch00]. We will propose a similar asynchronous algorithm for progressive radiosity. Moreover, effective dynamic load balancing techniques will be presented in order to minimize the idle times. In addition, we will present a novel algorithm with deferred synchronization, in order to solve technical difficulties of the asynchronous algorithm. This algorithm is not only efficient, but is also able to control the degree of synchronization.

4 Design of Algorithm

Before we present the parallel algorithms with different synchronization properties, we will introduce basic features of the environment we expect these algorithms to be running in.

4.1 Assumptions

4.1.1 Target Architecture

We will identify suitable architectures for our algorithms according to the classification of parallel architectures by [Cap93]. Presented algorithms are suitable for the MIMD computer architectures, with distributed memory organization. These architectures, known as multicomputers, are very common today. Our algorithms are asynchronous and try to exploit the asynchronous communication model. In order to utilize their power, asynchronous architectures need to be used.

Also, we assume that the processors used are of the same speed. If processors differ in speed, but their speeds are given, it is not hard to extend the proposed algorithms.

4.1.2 Communication Model

In our approach, we will use the message passing model used in distributed architectures. We will use four basic operations in the message passing framework, presented in [Pla06]:

Definition 10 (Submission of a basic message passing operation.)**.** *Submission of a basic operation denotes the act of passing the operation from a process to the message passing system.*

Definition 11 (Representation of basic message passing operations.)**.** *All basic message passing operations are tuples [op, x, Y, m, f, s, t], where op $\in$ CREATE, DESTROY, SEND, RECV; x is the identifier of the process which* *submits the operation; Y is a set of process identifiers; m is a message; f is a boolean function defined on messages (a filter); s is either a reference to a semaphore object which can be accessed by the message passing system, or NULL; t is the time stamp of the submission of the operation (i.e. the time when the operation has been read by the message passing system).*

The SEND operation is unblocked, while the RECV operation is blocked. In order to achieve possibility of an asynchronous communication, we will only be using a *communication thread* for submitting the RECV operation and treating the incoming messages. This is possible, since the framework is thread-safe.

We assume that all the processors are non-faulty. We also assume that the communication channels are non-faulty and that they maintain the order of messages — if a particular process sends two messages in a certain order to another processor, both the delivery and the order of messages are guaranteed.

We assume that all processors are fully connected and can communicate with each other. We do not take into consideration the physical topology. We work with the logical topology of full graph, since it can be simulated on all usual physical topologies. The problem of routing can be solved independently.

4.1.3 Memory Limitations and Termination Criteria

In all presented algorithms, we assume that the size of the input scene geometry is small enough to fit to memory of every processor. This is not very restrictive, since usual scenes have $\sim$ 100.000 patches[21] and very large scenes have $\sim$ 1.000.000 patches, what corresponds to approximately 500 MB of memory. Moreover, since the scene geometry is read-only, it can be easily stored in a distributed database with read-only access [GP89], [Gra98], [Pla03].

[21]Patch is a part of the scene surface, usually a polygon. It is a result of the discretisation of scene surface.

Considering the termination criteria, we have two options here. We may terminate the algorithm when total residual values are below certain threshold. However, we may also define a maximal residual allowed and terminate when all residua are less then equal to the allowed maximum. Both this criteria are equivalent, in the sense that we can assure the validity of one criterion by a corresponding setting of the other one. For parallel purposes however, the criterion defined as the greatest residual allowed is more suitable, because if the criterion is met for all parts of the scene individually, it is also met for the whole scene.

4.2 General Overview

Since any shading algorithm or ray tracing can be used for the second phase of radiosity method, we will focus on the first phase only. An interested reader can find out more about parallelization of ray tracing in e.g. [RCJ98] or [GP89]. In this chapter, we will present a simple synchronous parallel algorithm followed by a fully asynchronous and an algorithm with deferred synchronization. We will first give the reader a notion of our basic approach to radiosity computation used in all presented algorithms.

In our parallel algorithms, we will follow the progressive radiosity principle. We will use the Monte Carlo methods for form factor computation. For these computations, form factor estimator proposed in [Pie93] is very suitable. It is based on analytical computation of the inner integral of (25), with additional stochastic ray casting method to estimate the visibility conditions between patches.

Our basic idea is to follow the data parallel approach to exploit the parallelism at a low level. The scene will be divided among the processors, so that every processor will be assigned a local set of patches of its own. These local sets of patches are disjunct. Additionally, we will replicate the complete scene geometry in every processor's memory. Since the scene geometry data are read only, the case when the scene does not fit into memory can be handled independently, using a distributed database with read-only access [GP89], [Gra98], [Pla03]. We use the geometry information for form factor computation and we do not alter it during the algorithm. Every patch from a processor's local set represents a potential work for the processor. If the energy of a patch is high enough,[22] it will eventually have to be shot on all other patches, including other processors' local sets. Algorithms differ in the strategy of patch selection. However, we always try to select patch with large unshot energy, as we stick to the principle of greatest residual (as in the Southwell relaxation). Once selected, the patch actual state and a time-stamp is recorded.[23] We will refer to this unique record as to *shooter*. After being selected, shooter is sent to other processors to be shot at the patches in their individual local sets. In the computation, some patches may be selected more times to have their energy shot, but at different moments, therefore represented by different shooters. Eventually, every processor will shoot the energy of a particular shooter at all patches in its local set and will do it only once. This includes the computation of the form factors between the shooting patch and receiving patches and updating the values of local patches after the energy from the shooter is transferred. We should note, that the actual residual and total radiosity values related to a particular patch are stored only in the local set of the processor that this patch belongs to. The patches keep being selected, until all the patches in every local set satisfy the termination condition. Then the algorithm ends.

4.3 Simple Synchronous Algorithm

In order to better explain what we understand under asynchronous algorithm, we will present a simple synchronous algorithm based on progressive radiosity. Like the progressive radiosity, the algorithm will work in iterations, until

[22]Its unshot energy is smaller than the maximum allowed. This maximum is given as the termination criterion as addressed in section 3

[23]Since patch geometry stays the same during the computation, our main concern is to avoid the alteration of radiosity values, particularly the energy to be shot, because it might change locally over time and cause data inconsistencies in the asynchronous algorithm.

the global[24] termination criterion is met. Every iteration consists of several phases:

1. First, the global termination condition is checked. Every processor evaluates if convergence is met for its local set and sends the information to the master.[25] The master decides whether the global termination criterion is met or not and broadcasts the information to all other processors. In the first case, the algorithm will quit.

2. Next, the processors will globally select the patch containing the largest unshot energy in the following fashion: Every processor selects the strongest shooter from its local set of patches and sends it to the master. When the master receives all locally selected maxima, it selects the global maximum and broadcasts it to all other processors. Note, that this phase can be included in the first phase for optimization.

3. Having received the shooter for this iteration, particular processor will compute the form factors between the shooter and the individual patches in its local set. Then it transfers the corresponding amounts of energy to its local patches accordingly.

Because of different visibility conditions and different patch sets assigned, the third phase will take different times on the individual processors. The faster[26] processors will move on to the first phase of the next iteration sooner than the slower ones. The first phase is a synchronization phase, because the master will not test the situation against the termination criterion sooner than all the processors are done with the previous iteration. This implies that some processors will have to wait. To avoid this situation, we may try to use a loadbalancing technique such as work stealing. This will be

[24] Global means, that the condition will hold considering all the patches in all processors, not just the individual local sets

[25] Master processor is identified using the *rank* identifier. For instance it can be processor with *rank* zero. This processor is used to decide on common cooperative tasks.

[26] Those which finish the iteration sooner than other processors

discussed later in this chapter. A better way, in our opinion, is to avoid the synchronization.

```
Program Synchronous;

/*INPUT*/
  array E; /*light sources radiosities for all patches*/
  array P; /*patch geometry and reflectancy description
            *for all patches            */
  criterion; /*greatest residual allowed*/
  rank;      /*unique identifier        */

/*VARIABLES*/
  var array B; /*total radiosities of local patches*/
  var array R; /*residua of local patches*/
  var array QREF; /*indexes to patch geometries for patches
                   *in the queue of shooters                 */
  var array QRES; /*residua of patches in the queue of shooters*/

BEGIN
  candidate = Select_Shooter_Candidate_From_Local_Set();
  candidate_residual = R[candidate];
  Send(to_master, "SHOOTER CANDIDATE",
        candidate, candidate_residual);
  if ( rank == 0 ) /* master selects among candidates */
  {
    Receive_Candidates_From_All_Processors();
    shooter = Select_Candidate_With_Greatest_Residual();
    if ( shooter_residual > criterion )
    {
    Broadcast("SHOOTER", shooter, shooter_residual);
    }
    else
    {
    Broadcast("END");
    }
  }
  Receive(msg);   /* synchronization */
  while( msg.TAG != "END" )
  {
    if ( msg.TAG = "SHOOTER")
    {
      shooter = Unpack_Shooter(msg);
      shooter_residual = Unpack_Residual(msg);
      for all i from local set
      {
        F = Compute_Form_Factor(P[shooter], P[i]);
        increase = r[shooter] * F * shooter_residual *
                   * A[shooter] / A[i];
        E[i] := E[i] + increase;
        R[i] := R[i] + increase;
      }
      if P[shooter] in local set
      {
        R[shooter] := 0;
      }
    }
    Dispatch(msg);
    candidate = Select_Shooter_Candidate_From_Local_Set();
    candidate_residual = R[candidate];
    Send(to_master, "SHOOTER CANDIDATE",
         candidate, candidate_residual);
    Receive(msg);  /* synchronization */
  }
  Dispatch(msg);
END
```

4.4 Fully Asynchronous Algorithm

In order to avoid waiting in the synchronization phase, we could leave out the synchronization phase and let the faster processors start another iteration. We allow an arbitrary processor to start an iteration and select the patch — shooter — to have its energy shot. In fact, several iterations can be run at the same time. We will ensure that all the iterations will finish eventually. Iterations may be executed in different order on different processors, but the data used will be the same as when the iteration started, since we "freeze" every shooter's data upon selection. This shooter will be delivered to all processors, stored in a *queue of shooters* and processed eventually. Therefore, the iteration will finish eventually. This approach brings us to the fully asynchronous algorithm — Fig. 4.

As we have already mentioned, processors are allowed to start an iteration without synchronization. The question arises, which patch should be selected by a processor for another iteration. We would like to stick to the progressive radiosity principle and choose the greatest residual. However, every processor has only information about the residua related to its local patches. In such a case, the choice of the globally greatest residual will be most likely violated. Anyways, we will choose the greatest residual *available* at the moment. Note, that processors may choose to continue an iteration started by another processor (if any, there will be shooters waiting in the queue of shooters), instead of starting a new one. In that case, we will again decide for the largest residual — we pick the best candidate from the local set of patches and we pick the best candidate from the queue of shooters. Finally, we choose the one with the greatest residual. If there is a strong residual in the local set, it will be soon selected and sent to others. In this fashion, mostly strong shooters will be used first and the algorithm will be very similar to the progressive radiosity algorithm.

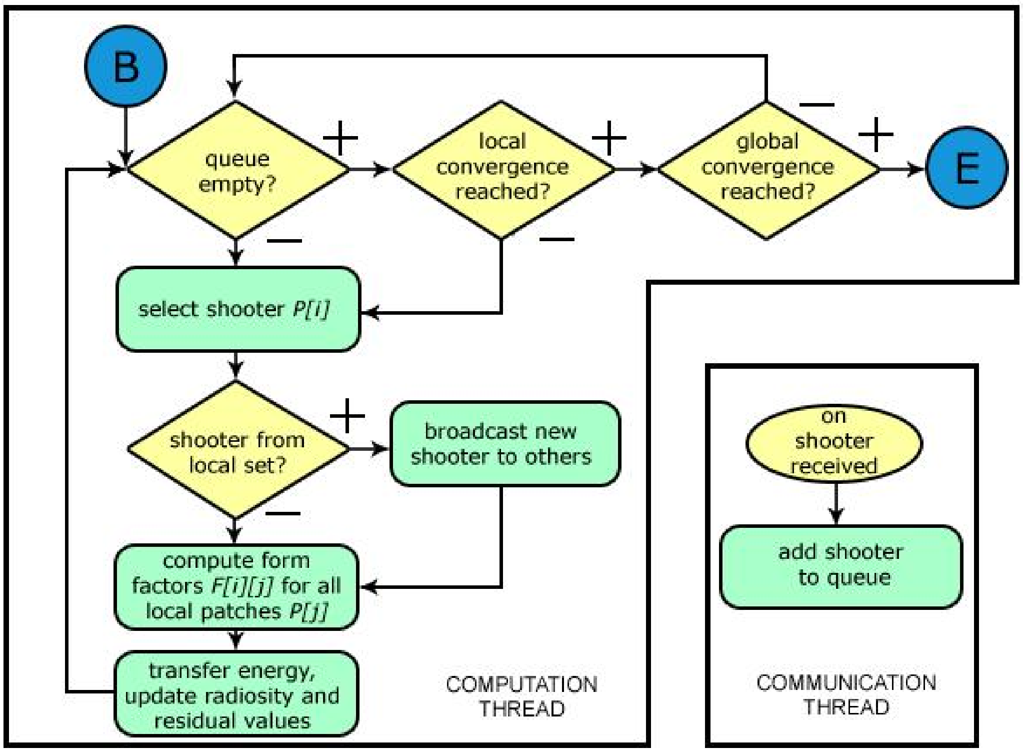

B
queue empty?
+
−
local convergence reached?
+
−
global convergence reached?
+
−
E
select shooter *P[i]*
shooter from local set?
+
−
broadcast new shooter to others
compute form factors *F[i][j]* for all local patches *P[j]*
transfer energy, update radiosity and residual values
COMPUTATION THREAD
on shooter received
add shooter to queue
COMMUNICATION THREAD

Figure 4: Basic asynchronous algorithm with two threads. Patch and geometry distribution can be done in the pre-computation phase.

In order to receive a shooter incoming from another processor, the receiver must run the *Receive()* procedure. The call of *Receive()* blocks the current execution until a message is received. In a sense, this is synchronization. The receiver must wait for a message to accept — it must wait for someone to reach the *Send()* procedure. Such synchronization is used in the simple synchronous algorithm, where the processors wait for the global shooter. In the asynchronous algorithm however, there are no global synchronous iterations and it is not known, when there will be message with a shooter coming and if there will be any at all. Therefore we can't use the blocking *Receive()* call. To solve this issue, our algorithm uses two threads. One thread is meant for computation, another one for communication. The second thread waits for a message to receive, while the first one is free to work. For better explanation, we can use an *worker/manager* analogy here. The computing thread is the worker and does the computation needed, where the communication thread is the manager who supervises the worker and assigns work if any.

In our case, computation thread computes the iterations, selects new shooters and sends them to all other processors, and checks the queue for received shooters. The communication thread receives messages with packed shooters and adds them to the queue. Note, that both threads are accessing the queue, therefore, we avoid the concurrent access by using a semaphore. The algorithm follows.

```
Program Asynchronous;

/*INPUT*/
  array E; /*light sources radiosities for all patches */
  array P; /*patch geometry and reflectancy description
            *of all patches*/
  criterion; /*greatest residual allowed*/

/*VARIABLES*/
  var array B; /*total radiosities of local patches*/
  var array R; /*residua of local patches*/
  var array QREF; /*indexes to patch geometries
                   *for patches in the queue of shooters*/
  var array QRES; /*residua of patches
                   *in the queue of shooters*/


COMPUTATIONAL THREAD BEGIN
```

```
  for all i from local set
  {
    B[i] := r[i] * E[i];
    R[i] := r[i] * E[i];
  }

  local_max = Select_Candidate_From_Local_Set();
  queue_max = Select_Candidate_From_Queue();  /* returns -1
                                                 if queue empty */

repeat {
  while( (R[local_max] > criterion) OR (queue not empty) )
  {
    if ( R[local_max] > QRES[queue_max] ) /* QRES[-1]
                 * is equal to -1 for the case of empty queue */
    {
      shooter = local_max;         /* index to patch geometry */
      shooter_residual = R[shooter];      /* residual */
      R[shooter] := 0;
      Broadcast("SHOOTER", shooter, shooter_residual);
                        /*freeze the residual and sent broadcast*/
    }
    else
    {
      shooter = QRES[queue_max];   /* index to patch geometry*/
      shooter_residual = QRES[queue_max];
      Lock_Queue;
      Remove_From_QRES(queue_max);
      Remove_From_QREF(queue_max);
      Unlock_Queue;
    }
    for all i from local set
    {
      FF = Compute_Form_Factor(P[shooter], P[i]);
      increase = P[shooter].reflectancy * FF *
                 * shooter_residual * A[shooter] / A[i];
      E[i] := E[i] + increase;
      R[i] := R[i] + increase;
    }
    local_max = Select_Candidate_From_Local_Set();
    queue_max = Select_Candidate_From_Queue();
  }
} until global convergence reached


COMPUTATIONAL THREAD END

COMMUNICATION THREAD BEGIN

Receive(msg);
  while( msg.TAG != "END" ) /* this message comes from the
  termination algorithm, which will be explained later
  {
    if ( msg.TAG = "SHOOTER")
    {
      global_shooter = Unpack_Shooter(msg);
      global_shooter_residual = Unpack_Residual(msg);
      Lock_Queue;
      Add_To_QREF(global_shooter);
      Add_To_QRES(global_shooter_residual);
      Unlock_Queue;
      Dispatch(msg);
      Receive(msg);
    }
  }
  Dispatch(msg);

COMMUNICATION THREAD END
```

For the sake of simplicity, details of the algorithm's termination were left out. In the algorithm shown, it is unclear where the computational thread has the knowledge about the global convergence from, since every processor can only check its own local set for the local termination criterion. Because of the choice of the criterion (maximal residual), once the iterations have finished and the criterion is met for all the local sets, also the global termination criterion is met. Unfortunately, termination in asynchronous approach is not straightforward. Although the local termination criterion may be met for all processors, there can still be some delayed message with a shooter on its way, as can be seen on Fig. 5. The energy of this shooter can invalidate the convergence criterion. Therefore, before terminating, we have to make sure there is no message hanging in the net. We will now describe how the algorithm terminates correctly. Basically, the termination is verified by the master processor. After that, the end is announced to every processor.

The worker's[27] role stays unchanged, except that it now informs its manager about the work status and the local convergence status. In order for the manager[28] to have the actual information about its worker's status, we introduce a shared variable — *idle*. The access to the shared variable is guarded by a semaphore. *Idle* is set to true by the worker in case that the worker has no work and is just checking the queue for incoming shooters (or waiting for work obtained by work stealing as will be discussed later). Note, that in the case the worker is idle, the local convergence holds, since otherwise the

[27] Computational thread
[28] Communication thread

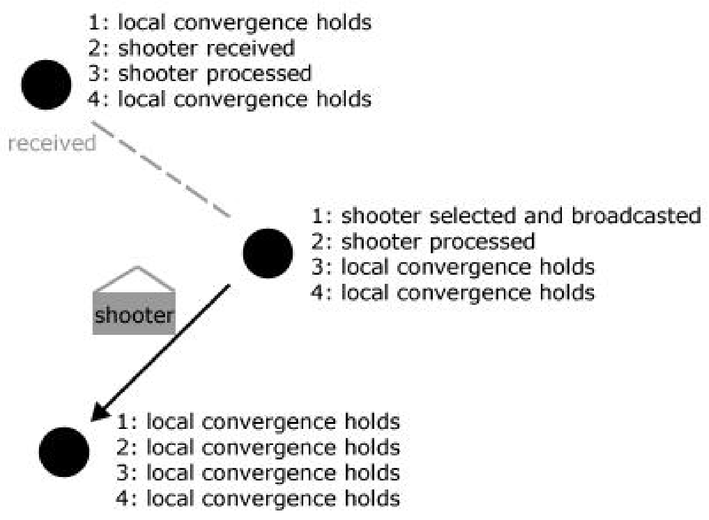

1: local convergence holds
2: shooter received
3: shooter processed
4: local convergence holds
received
1: shooter selected and broadcasted
2: shooter processed
3: local convergence holds
4: local convergence holds
shooter
1: local convergence holds
2: local convergence holds
3: local convergence holds
4: local convergence holds

Figure 5: Delayed shooter in the asynchronous algorithm prevents the global termination in time phase 4, even if the local convergence holds for all processors.

worker would select a shooter. Once a shooter is received by the manager, *idle* is set to false and the shooter is added to the queue. These two operations have to be carried out together, since the worker sets the *idle* variable according to the state of the queue. We have to ensure that the *idle* variable is always set properly. Therefore we perform necessary locking.[29] Once the worker has determined local convergence and the queue is empty, it will announce this to its manager — sends the `IDLE` message. After `IDLE` was sent, the manager knows that the local convergence holds. A `CONV FALSE` message is sent by the worker only when convergence has just been invalidated. Upon the receipt of convergence information from the worker (`IDLE` or `CONV FALSE` message), the manager announces the master manager, if there was a change (local convergence does not necessarily change when *idle* changes). For reference on all message types sent between threads and processors and used variables, see Fig. 6.

As we have mentioned, once a worker has determined local convergence and the queue is empty, it will announce this to its manager — sends `IDLE` message. The manager makes a note about the situation and sends a message `CONV TRUE` to the master manager. The master manager keeps a record of the local convergence state of all processors. Once the master manager thinks that all have reached local convergence, it runs the termination checking procedure. In the checking procedure it is determined, if there is something in the system that could prevent the global convergence. Once the termination checking procedure is started, the master manager switches to the **CHECK** mode and sends a `CHECK 0` message[30] to all managers. Upon receiving the `CHECK 0` message, every manager switches to the **CHECK** mode and performs following steps:

1. Sends a `PATH CHECK 0` message to all its neighbours.

[29]We use the same semaphore for controlling access to queue and the *idle* variable

[30]The number included in the message may be different, it depends on the actual value of the *attempt counter*. In case that this termination attempt fails, the next one's messages will have an number increased.

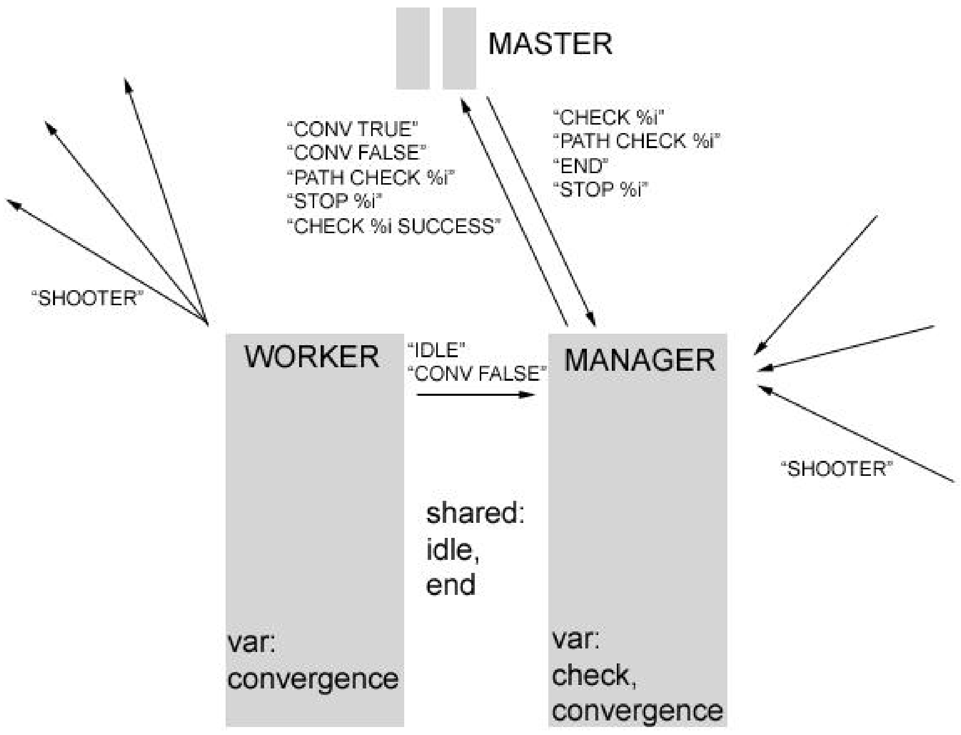


Figure 6: Message types used in the asynchronous algorithm.

2. The manager determines, if its worker's local convergence still holds. [31] When *idle* is set to true, local convergence holds. When *idle* is set to false, it is not certain, if the local convergence holds. This means that worker is processing a shooter, which might possibly invalidate the convergence status, but the local convergence check is yet to be done. Here we have to wait for the result and have two options. Either *idle* will be set to true by the worker, or the `CONV FALSE` message will be sent. From the communication thread concept — the need to receive messages (e.g. `CONV FALSE`), we cannot afford to keep checking the *idle* variable. But we can wait for the worker to send the `IDLE` message — the worker does it every time *idle* is set from false to true.

3. If the worker is idle, the manager awaits `PATH CHECK 0` messages from all neighbours.

Note, that the processing of steps 2 and possibly 1 is fired by receipt of a corresponding message. Therefore these steps may overlap. Once all messages from step 3 are collected and the local convergence still holds, the message `CHECK 0 SUCCESS` is sent to the master manager.

In case that the master manager receives successful checks from all managers, the global convergence criterion is met. The master manager broadcasts the termination announcement to all managers. The managers inform their workers by means of the shared variable *end*, and the algorithm ends. Note, that `PATH CHECK 0` message may arrive before the `CHECK 0` message. In that case, the receiving manager will switch to the **CHECK** mode and record, that the a `PATH CHECK 0` from one neighbour has been received already. While in the **CHECK** mode, only checking and termination messages and between-threads messages are considered safe. Once any other message is received, the termination was violated and the receiver sends `STOP 0` message to every manager, runs the *Stop()* procedure and only then processes

[31]This must be done, since we allow the worker to process a shooter even after the local convergence was announced to master manager and before the manager switched to **CHECK** mode.

the received message. The *Stop()* procedure includes switching the **CHECK** mode off, cleaning the convergence record (for the master manager only), increasing the counter of unsuccessful checking procedures to 1, checking the status of the worker and sending `CONV TRUE` message to the master manager if the worker is idle. Upon receiving the `STOP 0` message, every manager runs the *Stop()* procedure. Note, that once the counter was increased to 1, managers ignore any numbered message with the value smaller than that of the counter (these are `PATH CHECK 0, CHECK 0, STOP 0, CHECK 0 SUCCESS`). This is done in order to distinguish between individual termination checks, so that they do not interfere.[32] Note that the `STOP i` message is also broadcasted in case that manager finds out that its worker has made changes to the data since convergence announcement and the local convergence does not hold.

4.4.1 Proof of Convergence and Termination

Consider the case, when there are no `SHOOTER` messages in the system and all workers have set the *idle* variable to true. We will refer to this as the *final state*. The final state means that the global convergence has been reached and all iterations have been finished.

The proof idea is the following: We will prove that the computational part of the algorithm is running until the final state is reached. Also, once the final state is reached, the algorithm will terminate, otherwise it will continue computing.

Lemma 4.1. *No deadlocks appear in the algorithm.*

Proof. Note, that there is no circular waiting except the termination checking procedure. Therefore we will only describe the situation when the managers are in the **CHECK** mode.

[32]Delayed message `STOP` from previous unsuccessful termination check could abort the current termination check.

The manager is in the **CHECK** mode and is waiting for the worker. Unless a shooter is received by the manager — that means the termination checking procedure will be canceled by `STOP i` messages, worker will not be added any additional work. Therefore, the worker will either reach local convergence and send the `IDLE` message, or will send the `CONV FALSE` message what will cause the cancelation of termination checking procedure. If `IDLE` was received (or *idle* was set) manager awaits the `PATH CHECK i` messages. These are sent by every manager once it has switched to **CHECK** mode. If one manager switches to the **CHECK** mode, all managers eventually will, unless the termination checking procedure is canceled. Therefore, all `PATH CHECK i` will be received eventually. Since there is no waiting for other messages, the algorithm will continue in execution. □

Lemma 4.2. *The algorithm will reach the final state eventually.*

Proof. Consider total residua TR at the beginning of the algorithm and also given termination criterion, greatest residual allowed $MaxR$. Then, $MaxR$ is a constant fraction of TR — $\exists k > 0 : k * MaxR = TR$. Given the residual R_i of any shooter selected during computation, it only takes a constant time to every processor to process this shooter. Note, that it does not matter when the shooter is processed. Also note, that we only allow to select shooters if local convergence was not reached, therefore it holds for any such residual R_i that

$$\forall i : R_i \geq MaxR \tag{57}$$

Recall the λ coefficient from the proof of convergence for Southwell relaxation (47) which describes the fraction of shot energy which stays in the scene.[33] Since $0 \leq \lambda < 1$, there will be residua decrease after every step by at least $MaxR * (1 - \lambda)$. Therefore, in case we can perform $k * \frac{1}{1-\lambda}$ steps (there is enough shooters with residua above $MaxR$) the total residua will be 0, therefore also the greatest residual is zero and the convergence is met. If there is not enough shooters, i.e. all shooters have their residua smaller than $MaxR$

[33] The other energy is dissipated and will not “bounce back”.

and the convergence is met. There can not be more than $k * \frac{1}{1-\lambda} * N$ shooters send, therefore there will not be any `SHOOTER` messages eventually. □

Note, that we did not prove that the asynchronous algorithm will necessarily converge to zero in case we set the $MaxR$ to 0.

Lemma 4.3. *While the execution of the algorithm is not in final state, the algorithm will not terminate.*

Proof. Assume that the termination checking procedure has started, but the algorithm is not in the final state (either the local convergence on some processor is not met, or there is a `SHOOTER` message). The master manager sends the `CHECK i` message and will not confirm the termination until `CHECK i SUCCESS` messages return from all processors.

Assume there is worker which does not meet the local convergence criterion. This worker manager will find this out (either immediately or will wait for `CONV FALSE` message) and send `STOP i` to all others including master manager and the termination will be canceled.

Assume `CHECK i SUCCESS` message is received from A and no `STOP i` message was sent in this termination check. It means, that A's worker is idle, all other processors are in **CHECK** mode (since A received `PATH CHECK i` messages from them) and there is no message on any path to A — since A received `PATH CHECK i` messages from them. If there was any message, it was "pushed" by the `PATH CHECK i` message, because messages of the same channel are received in the same order as they have been sent. Should there be a message sent after the `PATH CHECK i` from X, it means that X is no longer in **CHECK** mode, therefore `STOP i` message was sent somewhere in the system what violates our assumption. Therefore, if there is a `SHOOTER` message somewhere on its way to X, after all managers reached the **CHECK** mode, it must have been sent before the sender reached the **CHECK** mode. Therefore, the `SHOOTER` message will be received by X before the `PATH CHECK i`, and the termination will be stopped, since master manager will receive `STOP i` instead of `CHECK i SUCCESS`. □

Lemma 4.4. *Once the final state (global convergence) was reached, the algorithm will terminate properly.*

Proof. In the final state, managers do not process any `SHOOTER` messages since otherwise *idle* would be set to false. Then it is straightforward that all started iterations were finished, since otherwise there would be `SHOOTER` messages in the system or there would be workers working, since our model permits message-system's or processors faults. The master manager always eventually receives a message `CONV TRUE` if the corresponding worker is idle.[34] If the final state is met, then the master's record will show local convergency for all workers eventually, or it already does. Therefore the termination checking procedure has already started or will eventually. If it started before the final state was reached, it could not be accomplished, unless the final state was reached in the meanwhile, as we have shown in previous proof. After the final state was reached, there will be next attempt eventually. The attempts do not interfere, since every attempt has its number. Therefore let us only consider the case, where the termination checking procedure starts by the time the final state has been reached already. In that case, all managers will reach the **CHECK** state eventually. Every manager will send `PATH CHECK i` message to every neighbour, thus will also receive one `PATH CHECK i` message from every neighbour. Since all workers are idle, all managers will send `CHECK i SUCCESS` to the master manager eventually. After the receipt of these messages, the master manager will broadcast the `END` message and the algorithm will terminate correctly. □

[34]Master's record of local convergence states is only changed either directly from the worker's manager, or when the termination attempt failed. In the latter case, the record is cleared, but managers with idle workers will update the record soon after failure, since every `CONV TRUE` message will be processed when the **CHECK** mode is off

4.5 Algorithm with Deferred Synchronization

We will present another algorithm, which is based on the fully asynchronous algorithm. First we will explain the reasons why at least "deferred" synchronization should be incorporated into the originally asynchronous algorithm.

4.5.1 Queue Size Problems

Consider the following situation in the asynchronous algorithm presented previously — processor A has such patches in its local set that it is quite easy to compute form factors related to these patches.[35] In such a case, A can finish local iteration with a particular shooter much faster than other processors. I.e. the other processors will compute at their pace and occasionally send shooter to A. Since A is fast in iterations, it will empty its queue of shooters quickly and flood the other processors with its own shooters. In this fashion, the size of another (a slow ones) processor's (B) queue can grow quickly. This is inconvenient from the point of implementation, because of the fact that there are memory limitations. In order to reserve enough memory for the program, we should be able to determine its memory requirements, or delimit them.

Another inconvenience in this scenario is, that A is forced, since its queue empties quickly, to choose new own shooters without any reference.[36] These shooters may have small residua, thus their contribution to the convergence process is insignificant. Unluckily, these insignificant shooters' energy also has to be transferred on other processors' patches, thus wasting their computational time.

[35]For example, all patches are faced away from most of other patches in the scene, including light sources

[36]In case there are shooters in its queue, the processor can decide, if its own shooters are "stronger" and, if not, decide rather for the shooters from queue.

4.5.2 Queue Size Delimitation

In the asynchronous algorithm, once B's queue reached the limit, there is no time to prevent others from sending additional shooters, since many shooters may be on their way already. We will have to avoid these shooters to be sent — every processor will remember the number of shooters sent by itself, which are residing in B's queue. In case of A, once a certain threshold for sent shooters is met, A will stop to select new shooters from the local set — we will refer to this as to *blocked state*, Fig. 7. Of course, the actual number of A's shooters in B's queue may be smaller than A remembers, since B may have processed some already. We will refer to the number of A's shooters in B's queue as to actual $A|B$ score.[37] If A is in blocked state, we would like B to inform A if actual $A|B$ score is smaller than the threshold. Here we have to be careful so that we do not misinform A:

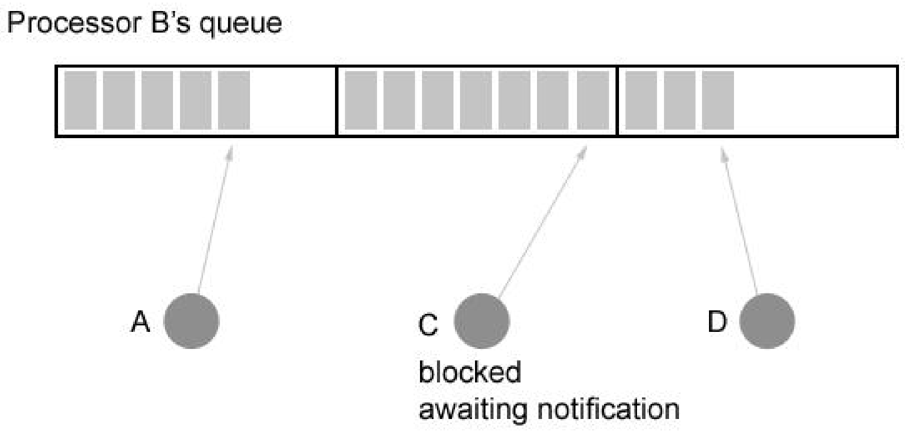


Figure 7: Processors are sending shooters and block themselves in case they reach the limit of queue space reserved for them.

Say that $threshold = 5$, actual $A|B$ is still 4, but A's record of $A|B$ is 5, since the fifth shooter has been sent recently. If B now informs A that actual

[37]Note that $A|B$ is different from $B|A$.

$A|B$ is 4, A will possibly send another shooter and exceed the threshold limit for B's queue.

From the previously shown reason, B informs A only if A is in a blocked state. To find out when A is blocked, B keeps record of what A thinks — B increases counter $count[A]$ every time shooter from A is received. Once the counter reaches the threshold value, B knows A is blocked. At that moment (or later), B may have already processed some shooters of A. Therefore it sends the notification — actual $A|B$ score to A[38], and assigns the $A|B$ value to the $count[A]$ counter in order to keep track of what A thinks. We may decide if B should notify A immediately after A has blocked itself and the actual $A|B$ score is under the threshold, or when the actual $A|B$ score dropped to a certain value — *trigger*.

Consider there are N processors. All processors other than B keep the score record of their own shooters sent to B. If we limit every processor's X/B score to *threshold*, the maximal number of all shooters (from different processors) in B's queue will be less or equal to $threshold * (N - 1)$.

4.5.3 Algorithm with Deferred Synchronization

Since A is blocking itself until notified by B, synchronization occurs in the algorithm. In comparison with the synchronous algorithm, the synchronization only takes place after few iterations.[39] Moreover, such synchronization only relates to a pair of processors. Therefore, we call this algorithm an algorithm with deferred synchronization.

In the algorithm with deferred synchronization, the threshold value is an interesting parameter, in the sense of amount of synchronization used. If we set the threshold value to 5, every pair of processors will always synchronize themselves at least after 5 "iterations".[40] Moreover, if we also set the *trigger*

[38]While A is blocked, it cannot influence the $A|B$ score.

[39]Determined by the threshold value.

[40]In the asynchronous algorithm, iteration is rather one shooting performed by particular processor.

parameter to 0, every pair of processors will always synchronize exactly after 5 "iterations". Therefore, when increasing the *threshold* parameter, we can defer the synchronization for several steps. If we set the threshold value high enough, we will get an algorithm very similar to the fully asynchronous one. On the other hand, if we limit the queue size to Q, threshold will be equal to $Q/N-1$. If there is a large number of processors, threshold will be relatively small.

We also solved the issue of insignificant shooters selection. Smaller thresholds bring the advantage of better shooter selection, since processors cannot compute "too-much-ahead" and shooting of insignificant shooters will not take place that often.

4.5.4 Algorithm Overview

As we have shown, the algorithm with deferred synchronization is a natural extension of the asynchronous algorithm. It also uses two threads for communication and computation. Considering the added functionality, these threads have to cooperate to achieve it. Basically, we can identify two additional independent groups of functionality in the extended algorithm from the view of the single processor A:

- **Functionality 1**: Control of the number of consecutively sent shooters to every particular processor B since the last received notification from this processor and blocking of shooter selection if necessary.
- **Functionality 2**: Recording of the number of received shooters from every particular processor X and sending notifications if the particular processor can be unblocked. This can be also influenced by the *trigger* value.

In the presented pseudo-code we will leave out the management of the shared variables and necessary communication from the asynchronous algorithm needed for proper termination. We will only show the added functionality. Also, we will not take the *trigger* values into consideration, since the algorithm can be easily extended for use of the *trigger*. The use of threads implies, that every shared variable access must be controlled — by a semaphore including both read and write access. If we do not state otherwise, we assume that every shared variable is locked before the immediate access and unlocked immediately after the operation on data is performed.

Considering the first functionality, the worker[41] only increases the $A|B$ score and blocks itself if the threshold is reached. Note, that it can still select and "shoot" the shooters from queue. After the worker was blocked, the possible incoming notification is then received by manager[42]. In order for the worker to start working, it must be informed by the manager that the notification has already arrived. It is sufficient for the manager to update the $A|B$ score according to received information — that certainly means lowering the score. We will use shared array *score* containing the $A|B$ scores[43] to solve this communication need. The worker reads the *score* values and determines if it should be in the blocked state.

Considering the second functionality, the manager records the number of received shooters from every particular processor X, increasing the $count[X]$ variable. The actual $X|B$ score can be influenced by both the worker and manager.[44] The worker decreases $X|B$ and the manager increases $X|B$. The notification is performed by both threads, depending on the situation:

- **Case:** By the time $count[X]$ reached the threshold, some of X's shooters stored in queue have been processed since last notification — actual $X|B < count[X]$.

  Manager: Upon receiving X's new shooter, it increases $count[X]$ and determines if the threshold was reached. If so, since $X|B < count[X]$ manager sends the notification and updates $count[X]$.

[41] Computation thread
[42] Communication thread
[43] However not the most actual ones.
[44] Note that actual $X|B$ score can be easily computed by counting corresponding shooters in the queue.

- **Case:** By the time $count[X]$ reached the threshold, none of X's shooters in queue have been processed since last notification — actual $A|B = count[X]$.
  Worker: Once shooter from queue is to be processed, worker determines if there is time for notification of the shooter's owner (possibly X), according to the $count[X]$ array. If so, worker sends the notification and updates $count[X]$.

Note, that the proper locking here is crucial. In both cases, the threads must have exclusive access to both $count[X]$ and queue. The locking order must be the same for both threads to avoid deadlocks.

```
Program Deferred;

/*INPUT*/
  threshold: int; /* a limit to the number of shooters allowed
                   * to sent consecutively to every particular
                   * processor without received notification */

/*VARIABLES*/
  var array score: int; /* for every processor this array
                         * stores number of shooters sent
                         * consecutively to that processor */
  var blocked: boolean;
  var array count: int; /* used for deferred synchronization,
                         * records if the number of received
                         * shooters from a particular
                         * processor reached threshold value*/

 COMPUTATIONAL THREAD BEGIN

 Do necessary initializations;
 for all i {
  count[i] := 0;
  score[i] := 0;
  end := false;
 }

repeat
  while( not converged OR queue not empty ) {
    for all i {                 /* we test for all processors*/
      if ( score[i] == threshold ) /* we are in blocked state*/
      {
        blocked := TRUE;
        break;
      }
    }
    if ( blocked )
      Select shooter from queue if there is one;
    else
      Select shooter from local set or from queue;

    if ( shooter selected from local set ) {
      Broadcast("SHOOTER");   /* freeze the residual
                               * and broadcast it */
      for all i {
        inc(score[i]);        /* the number of sent
                               * residuals increases */
      }
      Transfer shooter's energy on local set;
    }
    Lock queue;
    if ( shooter selected from queue )
    {
      Remove shooter from queue;
      j := shooter's owner;    /* the processor that has sent
                                * the particular shooter */
      Lock count[j];
      if ( count[j] == threshold )  /* if j-th processor
                                     * is blocked */
      {
        k := number of j-th processor's shooters in the queue;
        Send(to j, "SCORE", k); /* we notify j-th processor
                                 * about the actual score,
                                 * to unblock it */
      }
      Transfer shooter's energy on local set;
    }
  }
} until end == TRUE

COMPUTATIONAL THREAD END

COMMUNICATION THREAD BEGIN

Receive(msg);
  while( msg.TAG != "END" ) /* this is sent by the
                             * termination algorithm */
  {
    if ( msg.TAG = "SHOOTER")
    {
      j := shooter's owner;
      Lock queue;
      Lock count[j];
      Add shooter to queue;
      inc(count[j]);
      if ( count[j] == threshold ) {
        k := number of j-th processor's shooters in the queue;
        if ( k < threshold )
        {
          Send(to j, "SCORE", k);  /* we notify j-th processor
                                  * about the actual score,
                                  * to unblock it */
        }
      }
      Unlock count[j];
      Unlock queue;
      Dispatch(msg);
    }
    if ( msg.TAG = "SCORE")
    {
      j := msg.sender;
      score[j] := msg.score;
      Dispatch(msg);
    }
    .
    .   /* processing of other messages */
    .
  }
  end := TRUE;
  Dispatch(msg);

COMMUNICATION THREAD END
```

5 Loadbalancing Using Work Stealing

In the asynchronous algorithm, we have the patches divided among processors in the same fashion as in the synchronous algorithm. In fact, since the work to be done for a particular processor is to transfer energy on the local set, the amount of work done in a particular processor is similar in both algorithms.[45] As we have already mentioned, in the synchronous algorithm, there are idle times before every synchronization. In the Fig. 8, it can be seen that in the asynchronous algorithm, the idle times were just pushed down closer to the end of the algorithm. Consider a situation in the asynchronous algorithm, where a processor has reached the local convergence, its queue of shooters is empty. In case the global convergence was not reached yet, the processor is idle. There will be shooters coming from other processors sooner or later, but the processor cannot generate work on its own right now. In case the processor will only wait and process incoming shooters, we will likely not do any better than with the synchronous algorithm. Sometimes, the fully asynchronous algorithm may be faster — although the processors computational times differ in every iteration of the synchronous algorithm, they may "even-out" in the fully asynchronous algorithm,[46] in case the total work every processor did was the same. In case the total work spent by processors was not even, we just take all the relatively small idle times from the synchronous algorithm, move them down and connect together into a larger idle time.

In both algorithms, we would like to minimize the idle times somehow. For both algorithms, work equalization would be beneficial. Static loadbalancing may provide equal distribution of work. Note, that the synchronous algorithm is "sensitive" to complexity differences in the tasks[47] themselves, since there is only one task per iteration. The asynchronous algorithm is

[45] It is not equal, since the asynchronous algorithm does not always choose the greatest residual.

[46] That means they will finish at the same time and there will be no idle times.

[47] In this case a task is to transfer the energy of shooter on patches from local set.

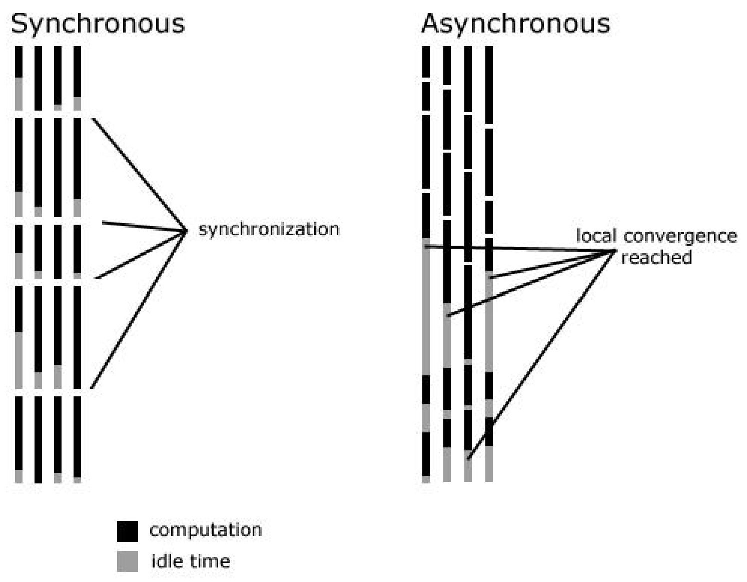


Figure 8: Comparison of idle times for the synchronous and asynchronous algorithm. Note, that in the asynchronous algorithm the idle times are compacted.

"sensitive" to the differences in the total amount of work done by individual processors. Therefore asynchronous algorithm should do better, using static loadbalancing only.

However, static loadbalancing is often not satisfactory, because imbalances may rise during computation itself (e.g. patch subdivision) and are hard to predict. To deal with this issue, dynamic loadbalancing is used. There are several approaches to dynamic loadbalancing. For the asynchronous message passing model, work stealing is very suitable.

5.1 Important Factors

The decision must be made, what kind of work should be "stolen" or transferred between processors, and how "fine" the tasks should be — task *granularity*. In the synchronous algorithm, imbalances have to be evened out at the iteration level, where a relatively small amount of work is done. Therefore the tasks should be even smaller. One form factor computation could be a task of desired granularity. However, in order to perform one stealing, communication is necessary. The communication time (latency) plays here an important role. In the synchronous algorithm, the idle times may be too short to perform work stealing. For example, processor A asks processor B for work, since A has finished the iteration. B sends A twenty form factor computations to be done. By the time B receives results from A, is B already idle, because the communication takes too long. Thus B could have finished earlier if it had computed those two form factors by itself. The smallest amount of work which can be efficiently stolen and performed is latency dependent. The finer the granularity of tasks, the better, since we can assign a more precise amount of them. Therefore, tasks of fine granularity are as well suitable for the asynchronous algorithm. However, asynchronous algorithm will be also resilient to larger-sized tasks, as well as to longer communication times. Moreover, considering large-sized tasks, quite a lot of work may be exchanged between processors in one task. Say that this corresponds to a hundred of tiny tasks. In case we can store the information corresponding to the large task in the same space as the small tasks, the data transfer of hundred tiny tasks is hundred times higher than transfer of one large task.

Who should we steal from? There are two radical options. To steal from all, or from one. For both cases, it is necessary to have the information of the current load in order to determine the amount of work to be stolen. Unfortunately, no global snapshot can be done about the processors' state. Also, any communication in order to gain additional information costs time and is therefore inappropriate. In the asynchronous algorithm, the *length of queue of shooters represents a good indicator of computational load.*

The next thing we should take into consideration, is the temporality of the "stolen" work.

- In the first case, the stolen work could be the determination of one form factor for example. This will put a temporary load on the target processor until the task is finished.

- In the second case, a processor could steal part of another processor's local set of patches. The stolen patches mainly represent a relatively large amount of possible future work, instead of a temporary job.[48] Also, it is hard to predict how much work are the patches related with. In a sense, this is similar to static loadbalancing.

Both these approaches have their own rationale. The choice of ideal work stealing is rather complicated and is out of scope of this thesis. We will present both above mentioned approaches in connection with two algorithms. However, our algorithms will be designed on a higher level and in an adaptable fashion, so that they can be linked with any work stealing algorithm.

5.2 General Work Stealing for the Asynchronous Algorithm

In the asynchronous algorithm, work stealing will be run only if the processor's queue is empty, and the local convergence was reached, therefore no work is available. In fact, this is also the right time to check if the global convergence was reached. In that case, instead of work stealing, we would rather end the computation and run a termination protocol. Before the presented termination algorithm starts, the worker sends "IDLE" message to its manager. The worker does so anytime there is no work. Thus, work stealing can be easily incorporated at this point of the algorithm. We only have to introduce another shared variables — *ws* flag (i.e. the manager is in work stealing state) and a list of stolen work — *tasks*. Once the manager receives the "IDLE" message, it sets *ws* flag to true, performs work stealing and after receiving of all work, passes it to the worker, setting *idle* to false and *ws* to false. While *ws* is set or task queue is not empty, the worker dedicates him-

[48]The work related with stolen patches lies in future shooters energy transfer onto these patches.

self to solving of those tasks.[49] Once the worker is done, possible results are sent to the corresponding computer. In case the worker is out of work again, "IDLE" message is sent again. Once the manager finds out there is nothing to steal,[50] it sends the "CONV TRUE" message[51] to the master manager in case the worker is still idle. In case there should be results sent back, the processors managers which have sent the work, have record of whom they have given work and what work. Upon receipt of the results, they clear the corresponding record.

While the work stealing is in progress — there are work stealing related messages, computing, or results are awaited — the termination procedure cannot succeed. Therefore in the check procedure the manager will have to check for the work stealing status too. Since the work stealing messages will abort the termination check automatically, it will be sufficient to verify if there is any record of to-be-returned results, if there is work awaited as a result of work stealing and if the worker is idle.

5.3 Iteration Stealing

In the *iteration stealing*, we consider a task to be equal to one local iteration (see Fig. 9), that means to transferring a particular's shooter energy onto patches from local set. Accordingly, the work we will steal is “shootings” on other processor's local set.

The work stealing proceeds as follows: First, a manager asks all other $N-1$ processors for work. In case there is a large number of processors we may only choose one to spare some resources. This choice is rather complicated and may include combination of approaches such as queue size heuristics, gathering information, or probabilistic methods. Every asked processor's manager will carry out the queue locking, determine the number of shooters

[49]In some cases we may also allow the worker to check if there is new work of its own. In that case, it returns the work back to the “lender”.

[50]The answers to work stealing requests contain no work.

[51]The manager does so upon receipt of last empty work stealing answer.

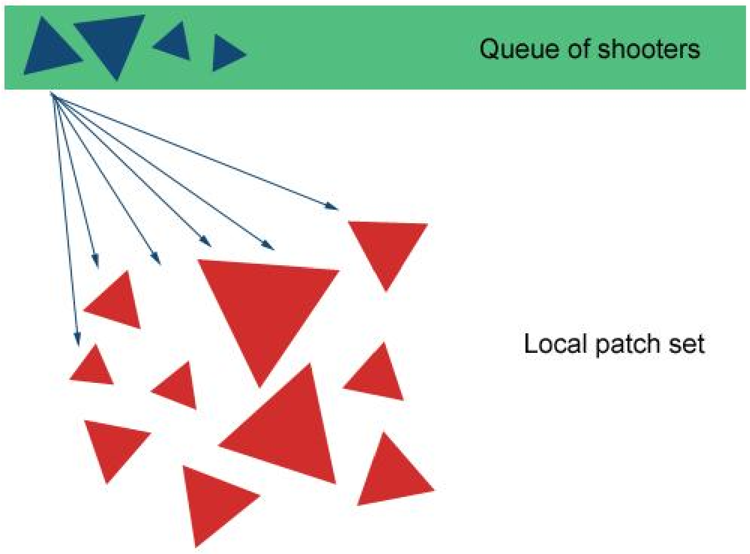


Figure 9: Example of one local iteration. Energy of queue shooter is transferred onto the local set.

in the queue — $|Q|$ and return $\lfloor |Q|/N \rfloor$ of the "weakest" shooters[52] (Fig. 10), plus it will decide with probability $(|Q|/N) - \lfloor |Q|/N \rfloor$ to send one additional shooter.[53] After the manager has received this work, it hands it over to the worker. We assume that the scene also contains patch owner information (i.e., each patch "knows" which process it belongs to).

In this case, we can carry out the iteration, since we have the knowledge of the shooter's owner local patch set. Note, that the local sets do not change during the computation.

In case we cannot assure the manager the knowledge of the "lender's" local set, the lender also appends the identification of its local set to the message with the stolen shooter. Consecutively, this data are received by the applicant's manager, and inserted into the task queue. The worker proceeds with the computation and sends the corresponding radiosity and residua

[52] Which correspond to the less important iterations.
[53] This solves the situation with larger amount of processors and little work.

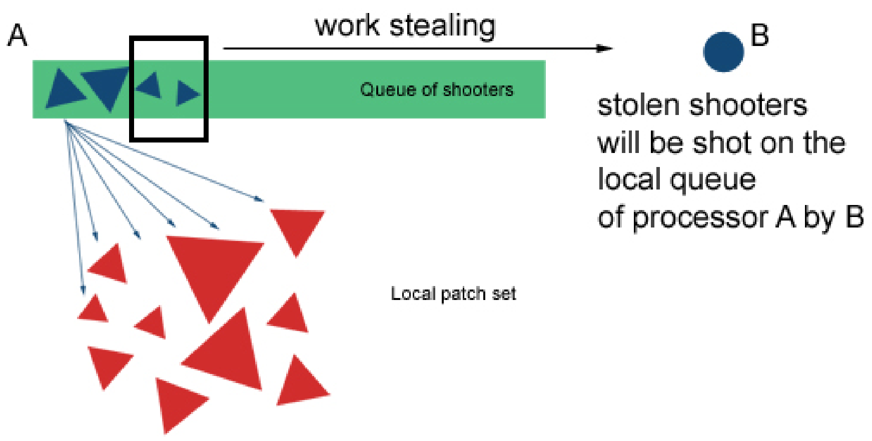


Figure 10: Iteration stealing. Work is contained in the shooters to be shot.

results to the shooter's owner. The owner's manager receives the results and updates the residua of the local set. Note, that locking is necessary, since common access is required. After the worker has finished processing the stolen work, it proceeds with the usual algorithm.

This approach has the disadvantage that relatively large amounts of data have to be transferred in order to deliver the computed results of stolen work, if processor's local patch set contains a lot of patches — say 10.000.[54] We can identify tasks at a lower-level — at the level of form factors. However, there would be additional control and synchronization necessary.

The advantage of this work stealing approach is, that the amount of work contained in a task can be estimated quite accurately. Also, since we steal a temporary work, the load balancing can be applied at arbitrary point in the algorithm.

[54]Then, 10.000 residua (results) must be sent after only one task was stolen.

5.4 Patch Stealing

We will present another approach based on work stealing. In this algorithm, we will rather "steal" data than work, still there will be work hidden in relation to the data. The principle can be seen on Fig. 11.

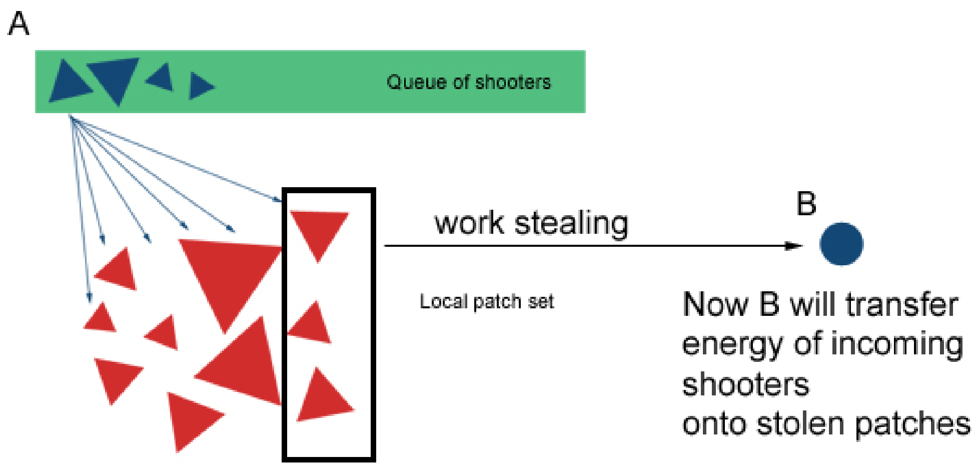


Figure 11: Patch stealing. Work is contained in the stolen patches to (possibly) have energy shot onto.

The patch stealing procedure works on the same principle as the general work stealing. We have already mentioned, that we will steal a part of some processor's local set. Since there are complications with the data consistency as we shall show, it is wiser to choose only one processor to steal from. In order to make the right choice, the particular processor A will at first ask all the other processors about their speed — the number of total iterations computed — I. Afterwards, the processor B with highest load is selected. The asked processor will sent a certain part of its data to the "applicant" A. Afterwards, A will unite the received patches together with its local set.

There are two problems to be discussed:

- to determine the amount of work to be "stolen"
- to keep the data in a consistent state

The first problem is how to determine the number of patches to be sent in response to the work request. If every processor remembers the total number of form factors computed — F, we can approximately compare its speed with others. The applicant will send F count and as well patch count P[55] to the particular "lender", which can compare their relative speeds.

We would like the processors to have the same number of iterations performed from the beginning. Since the number of iterations can be expressed as F/P we have

$$\frac{F[A]}{P[A]+x} = \frac{F[B]}{P[B]-x} \tag{58}$$

where x is the number of to-be-sent patches. From this we have

$$x = \frac{F[A]P[B] - F[B]P[A]}{F[A]+F[B]} \tag{59}$$

After this procedure, B and A update the number of finished iterations: $I'[A] = I'[B] = (I[A] + I[B]\ )\ /\ 2;$

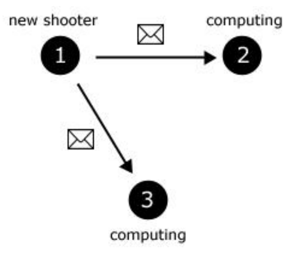
Another problem is connected with data consistency. First, we have to pay attention when removing the data from the lender's local set. There should be no iteration going on when the patches are extracted, since if there is iteration running, the shooter's could be shot on some patches only and others may stay left out.[56] If we have extracted our shooters, we send them and attach the actual queue of shooters, because these shooters' energy has to be shot on the sent patches too.

Then there is another problem which can show up when the patches with queue are on their way. They may miss an incoming shooter as shown in Fig. 12.

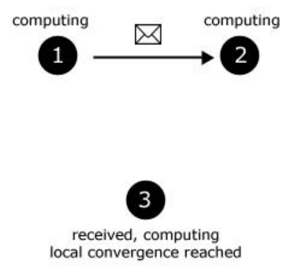
In order to solve this issue, we will have to store in every processor the information of last received shooter from every processor — S. If A asks B

[55]Number of patches in its local set

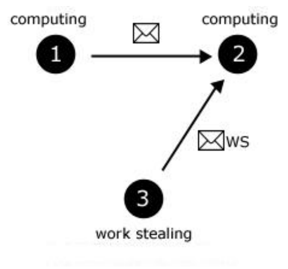
[56]This can be optimized so, that we divide the local set into two parts and we remember which part is processed actually. Then we select the patches from the inactive half.



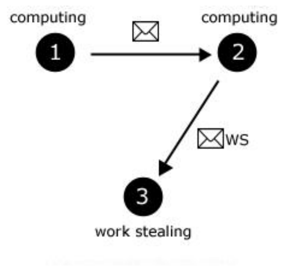
(a) Processor 1 sends new shooter



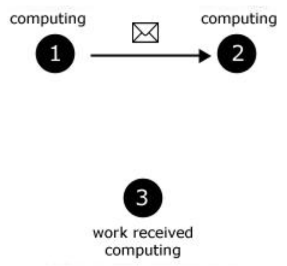
(b) Shooter received by processor 3 only



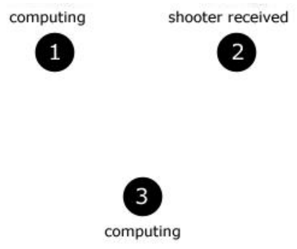
(c) Processor 3 asks processor 2 for work



(d) Proc.3 receives work form proc.2



(e) Shooter message still hanging



(f) Shooter received by proc.2

Figure 12: Data consistency problem. Shooter misses the patches which are not stored in any of the processors, instead they are somewhere in net.

for patches, the worker blocks itself, the $S[A]$ is recorded and sent to B. No incoming shooters are processed until the work stealing is complete. Since the shooters always have a time stamp, we can compare them to shooters which originated in the same processor.

Definition 12. *We say that record of last received shooters* $S = "s_1, s_2, \ldots, s_n"$ *is greater or equal than* $R = "r_1, r_2, \ldots, r_n"$*, if* $\forall i : s_i \geq r_i$*. We similarly define the other relations.*

After B received the request from A with $S[A]$, it compares $S[A]$ with $S[B]$. $S[B]$ must be greater or equal than $S[A]$ in order to select the patches — if some $S[A][i]$ is greater than the corresponding $S[B][i]$, that means A has received some shooter which B has not. Therefore B must wait for this shooter so that it does not miss the to-be-stolen patches, instead to be added into the queue. Once this condition is fulfilled, patches can be sent. In order for A to unite its local set with the received set, it must unite the queues related with these sets. Since $S[B] \geq S[A]$ and A was not processing any shooters, now B can wait for and process necessary shooters such that eventually $S[B] = S[A]$. Afterwards the sets can be united.

An advantage of the patch stealing is that the data transfer induced by patch transfer and other necessary data is relatively small. Also, we do not have to send any results. However the amount of work caused by the individual patches is hard to predict. Patch stealing is not suitable for the asynchronous algorithm, where work stealing is done near the end of computation, when stolen patches do not represent any significant load. Moreover, the implementation brings difficulties with data consistency, since the data are moved between processors.

5.5 Work Stealing for the Algorithm with Deferred Synchronization

There are two points in the algorithm with deferred synchronization where there are idle times:

- The processor has sent too many shooters, thus is blocked and awaits notification.
- The processor reached local convergence and there is no work.

The second case is actually the same as by the asynchronous algorithm. Therefore, iteration stealing should be used to obtain new work.

In the first case, we have two options. Even though the patch stealing is not suitable for the asynchronous algorithm, it can be used at this point in the algorithm with deferred synchronization, because the patch distribution significantly influences the load on the processors.

The iteration stealing is also suitable for the point in the algorithm when a certain processor is blocked. Note, that at this point the processor may decide to process shooters from its queue. However, this may not be optimal because it may violate the assumptions of using the greatest residual available. If we want to use iteration stealing, it is reasonable to steal work from the processor, which prevents us from computing. This is a slight modification in comparison with the approach we have presented. The blocked processor only asks the blocking one for shooters (related to iterations). The blocking processor returns $|Q|/N$ shooters, where $|Q|$ is the number of shooters in the queue and N is the number of processors. Otherwise the algorithm is the same as the iteration stealing presented before. However, there is a place for optimization in the case of algorithm with deferred synchronization. For A, instead of waiting in the blocked state, A continues in the computation. In case a new shooter is selected from the local set, instead of sending the selected shooter to every processor, we only send it to the nonblocking ones. A computes the shooter's energy transfer on its local set, but also on the sets of the blocking processors. This is possible, because we assume that the distribution of patches between processors is known to every processor and that it does not change.

6 Conclusion

In this thesis, we presented two efficient parallel algorithms based on the progressive radiosity method. These algorithms were designed on a high level. In connection with the asynchronous communication model and separate communicationthread, this implies small communication demands. Both the algorithms are not only efficient, but also very flexible. Our algorithms do not require any specific partitioning of the scene geometry. They can be viewed as a parallel framework and can be easily connected with different sequential algorithms based on progressive radiosity.

The presented asynchronous algorithm computes the radiosity solution with perfectly minimized communication, while it retains a high degree of convergence rate. In comparison with the simple synchronous algorithm, the asynchronous algorithm is more resistant to load imbalances and can be very easily combined with work stealing techniques to obtain high speedups. We have proved that the algorithm converges for a given accuracy to the radiosity solution.

In addition, a novel efficient algorithm was proposed — the algorithm with deferred synchronization allows us to control the degree of synchronization, and controls the amount of memory used during computation. If there is enough memory available, the algorithm has the same properties as the asynchronous algorithm.

For further improvements, we proposed two different approaches to work stealing. *Iteration stealing* is a very general method for dynamic load balancing and can be used in all situations. *Patch stealing* is a method of dynamic loadbalancing based on data transfer instead of work transfer. It is very similar to static load balancing in the sense that it influences the load for the rest of computation. The amount of data transferred using patch stealing is independent on the corresponding work.

References

[APRP96] Bruno Arnaldi, Thierry Priol, Luc Renambot, and Xavier Pueyo. Visibility Masks for Solving Complex Radiosity Computations on Multiprocessors. In *Proc. First Eurographics Workshop on Parallel Graphics and Visualisation*, pages 219–232, Bristol, UK, 1996.

[Bek99] Philippe Bekaert. *Hierarchical and Stochastic Algorithms for Radiosity*. PhD thesis, Leuven, Belgium, 1999.

[BFGS86] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. Image rendering by adaptive refinement. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 29–37, New York, NY, USA, 1986. ACM Press.

[BP94] K. Bouatouch and T. Priol. Data management scheme for parallel radiosity. *Computer Aided Design*, 26(12):876–882, 1994.

[BW90] Daniel R. Baum and James M. Winget. Real time radiosity through parallel processing and hardware acceleration. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 67–75, New York, NY, USA, 1990. ACM Press.

[Cap93] Tolga K. Capin. Parallel Processing for Progressive Refinement Radiosity. Master's thesis, Ankara, Turkey, 1993.

[CCWG88] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 75–84, New York, NY, USA, 1988. ACM Press.

[CG85] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: a radiosity solution for complex environments. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 31–40, New York, NY, USA, 1985. ACM Press.

[CWH93] Michael F. Cohen, John Wallace, and Pat Hanrahan. *Radiosity and realistic image synthesis*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[DS92] Steven M. Drucker and Peter Schroder. Fast radiosity using a data parallel architecture. In *Third Eurographics Workshop on Rendering*, pages 247–258, Bristol, UK, 1992.

[Fly66] M.J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE, Volume: 54, Issue: 12*, pages 1901–1909, 1966.

[FP91] Martin Feda and Werner Purgathofer. Progressive refinement on a transputer network. In *Proceedings of 2nd Eurographics Workshop on Rendering*, pages TALK: M. Feda.

[Fun96] Thomas A. Funkhouser. Coarse-grained parallelism for hierarchical radiosity using group iterative methods. *Computer Graphics*, 30(Annual Conference Series):343–352, 1996.

[GCS93] Steven J. Gortler, Michael F. Cohen, and Phillipp Slusallek. Radiosity and Relaxation Methods: Progressive Refinement is Southwell Relaxation. Technical Report CS-TR-408-93, Princeton, NJ, 1993.

[Gla89] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., London, UK, UK, 1989.

[GP89] S.A. Green and D.J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, 5(6), 1989.

[Gra98] Pavol Gramblička. Cache techniky pre paralelný raytracing. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia, 1998.

[GRS95] Pascal Guitton, Jean Roman, and Gilles Subrenat. A parallel method for progressive radiosity. Technical Report Research Report 992-95, Talence, France, 1995.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 213–222, New York, NY, USA, 1984. ACM Press.

[How82] J.R. Howell. *A Catalog of Radiation Configuration Factors.* McGraw-Hill, 1982.

[Šin95] Libor Šindlar. Paralelní algoritmz počítačové grafiky. Master's thesis, Matematicko-fyzikální fakulta Univerzity Karlovy, Praha, Czech Republic, 1995.

[Kel96] Alexander Keller. Quasi-Monte Carlo Radiosity. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 101–110, New York, NY, 1996. Springer-Verlag/Wien.

[Mal88] Thomas J. V. Malley. A Shading Method for Computer Generated Images. M.Sc. thesis, June 1988.

[Pie93] Georg Pietrek. Fast Calculation of Accurate Formfactors. In *Fourth Eurographics Workshop on Rendering*, number Series EG 93 RW, pages 201–220, Paris, France, 1993.

[Pla03] Tomáš Plachetka. *Event-driven message passing and parallel simulation of global illumination.* PhD thesis, University of Paderborn, 2003.

[Pla06] Tomas Plachetka. Unifying framework for message passing. In Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors, *SOFSEM*, volume 3831 of *Lecture Notes in Computer Science*, pages 451–460. Springer, 2006.

[PT89] M. Price and G. Truman. Radiosity in parallel. In *Proceedings of the 1rst conference on applications of transputers*, pages 40–47, 1989.

[PZ90] Werner Purgathofer and Michael Zeiller. Fast radiosity by parallelization. In *Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 173–183, June 1990.

[RCJ98] Erik Reinhard, Alan Chalmers, and Frederik W. Jansen. Overview of parallel photo-realistic graphics. Technical Report CS-EXT-1998-147, 1, 1998.

[RGG90] Rodney J. Recker, David W. George, and Donald P. Greenberg. Acceleration techniques for progressive refinement radiosity. *SIGGRAPH Comput. Graph.*, 24(2):59–66, 1990.

[Sch00] Olaf Schmidt. *Parallele Simulation der globalen Beleuchtung in komplexen Architekturmodellen.* PhD thesis, University of Paderborn, 2000.

[SP89] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 335–344, New York, NY, USA, 1989. ACM Press.

[SW] W. Sturzlinger and C. Wild. Parallel prograssive radiosity with parallel visibility computations.

[WCG87] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 311–320, New York, NY, USA, 1987. ACM Press.

[WEH89] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A ray tracing algorithm for progressive radiosity. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 315–324, New York, NY, USA, 1989. ACM Press.

[YIY97] Yizhou Yu, Oscar H. Ibarra, and Tao Yang. Parallel progressive radiosity with adaptive meshing. *Journal of Parallel and Distributed Computing*, 42(1):30–41, 1997.

[ZBSF04] Jiří Zára, Bedřich Beneš, Jiří Sochor, and Petr Felkel. *Moderní počítačová grafika.* Computer Press, Brno, Czech Republic, 2004.

Abstrakt

Táto práca sa zaoberá návrhom efektívneho paralelného algoritmu pre riešenie metódy radiosity. V paralelnom algoritme založenom na progresívnej radiosity používame asynchrónny model posielania správ. Vďaka nezávislému toku riadenia určenému pre prijímanie správ (komunikačné vlákno) môže komunikácia medzi procesmi prebiehať aj počas výpočtu. Tento prístup umožňuje modifikovať numerické riešenie problému, čím sa zabezpečí zníženie komunikácie medzi procesmi. Vďaka tomu je algoritmus efektívny. Výpočet je paralelizovaný v najvyššej možnej vrstve sekvenčného algoritmu, čo spôsobuje ďalšie zníženie komunikácie a tiež umožňuje vytvorenie paralelného frameworku nezávislého na implementácii nižších vrstiev. V práci je navrhnutý plne asynchrónny a čiastočne synchrónny prístup pre riešenie progresívnej radiosity. Pre porovnanie uvádzame aj tradičný synchrónny prístup. Práca tiež obsahuje návrh pre vyrovnávanie nerovnomerností záťaže, ktoré môžu nastať počas výpočtu.

Kľúčové slová: progresívna radiosity, paralelizácia, asynchrónny model