PARALLEL IMPLEMENTATION OF STOCHASTIC ITERATION ALGORITHMS

Roel Martínez, László Szirmay-Kalos, Mateu Sbert, Ali Mohamed Abbas

Department of Informatics and Applied Mathematics, University of Girona
Department of Control Engineering and Information Technology TU of Budapest,
(roel,mateu)@ima.udg.es, szirmay@iit.bme.hu

ABSTRACT

The paper examines the parallel implementation of iteration type global illumination algorithms. The steps of iteration depend on each other, thus their parallel implementation is not as straightforward as for random walks. This paper solves the interdependency problem by applying stochastic iteration. In this framework two fundamental questions are investigated: how many processors can be efficiently used in an algorithm and how often the processors should exchange their information. These questions are answered by a theoretical model and also by simulations.

Keywords: Global illumination, parallel computing, Monte-Carlo methods, radiosity

1 Introduction

Global illumination algorithms, which aim at the physically correct simulation of the light propagation, solve the rendering equation

$$L = L^e + \mathcal{T}L,$$

which expresses the radiance $L(\vec{x}, \omega)$ of a point $\vec{x}$ at a direction ω as a sum of the emission and the reflection of all point radiances that are visible from here. The reflection of the visible points is expressed by an integral operator $\mathcal{T}L(\vec{x}, \omega)$ which is also called as the light transport operator.

Global illumination algorithms can be classified as random-walk and iteration techniques. *Random walk* algorithms are based on the Neumann series expansion of the rendering equation

$$L = \sum_{i=0}^{\infty} \mathcal{T}^i L^e$$

where $\mathcal{T}$ is the light transport operator and L^e is the emission. The terms have intuitive meaning since $\mathcal{T}^i L^e$ describes the i-bounce transfer. The terms of this series are ever increasing high-dimensional integrals that are estimated by Monte-Carlo quadrature in order to avoid exponential core. Since the samples in the integral quadrature are computed independently, this method may be executed on as many processors as samples are needed. If the samples are available, then the results should be joined by the combination of the independent samples. Summarizing, random-walk algorithms are the optimal candidates for parallelization since only the final combination requires some inter-process communication. This feature, which is an obvious advantage for parallelization, is also a drawback, because it shows the inherent property of random walks that they are unable to reuse previous information. *Iteration* techniques, on the other hand, realize that the solution of the rendering equation is the fixed point of the following iteration

$$L(m) = L^e + \mathcal{T}L(m-1),$$

thus the radiance can be obtained as a limiting value:

$$L = \lim_{m\to\infty} L(m).$$

To store the radiance estimates during iteration, finite-element approaches should be used. Since the complete radiance function is used in each step, iteration can potentially reuse coherence and previous information, thus it can be much faster than random walks. On the other hand, due to the dependence on the complete radiance function iteration is not as appropriate for parallelization as random walk [6, 2]. If different parts of the radiance function is computed on different processors, then their results should be combined after each iteration (or at least after every few iterations), which requires intensive inter-process communication and can result in a bottleneck. This approach is feasible if the decomposition takes advantage of the separation of strongly and weakly coupled regions, such as, for example, the rooms inside a bigger building [3, 5].

These problems can be solved by randomizing the iteration. The formal basis of the randomization of iteration methods is the stochastic iteration [7]. It means that in the iteration sequence a random transport operator is used instead of the light-transport operator, which gives back the light-transport operator in the average case:

$$L'(m) = L^e + \mathcal{T}^* L(m-1), \quad E[\mathcal{T}^* L] = \mathcal{T} L$$

This scheme does not converge but $L'(m)$ will fluctuate around the real solution. To find the converged solution, before advancing to the subsequent step, the radiance is obtained as the average of all preceding radiance estimates:

$$L(m) = \frac{1}{m}\cdot\sum_{i=1}^{m} L'(i) =$$

$$\frac{1}{m}\cdot L'(m) + \left(1-\frac{1}{m}\right)\cdot L(m-1).$$

Merging the stochastic iteration and averaging steps together, we can also obtain:

$$L(m) =$$

$$\frac{1}{m}\cdot(L^e + \mathcal{T}^* L(m-1)) + \left(1-\frac{1}{m}\right)\cdot L(m-1).$$

In order to understand how iteration works, let us examine its first few steps with the assumption that $L(0) = L^e$:

$$\begin{aligned}
L(0) &= L^e, \\
L(1) &= \frac{1}{1}\cdot(L^e + \mathcal{T}_1^* L(0)) + \left(1-\frac{1}{1}\right)\cdot L(0) \\
&= L^e + \mathcal{T}_1^* L^e, \\
L(2) &= \frac{1}{2}\cdot(L^e + \mathcal{T}_2^* L(1)) + \left(1-\frac{1}{2}\right)\cdot L(1) \\
&= L^e + \frac{1}{2}(\mathcal{T}_1^* L^e + \mathcal{T}_2^* L^e) + \frac{1}{2}\mathcal{T}_2^*\mathcal{T}_1^* L^e.
\end{aligned}$$

Note that each iteration introduces a new one-bounce transfer $\mathcal{T}_i^* L^e$ and advances all previous transfers, i.e. obtains a $k+1$ bounce transfer $\mathcal{T}_i^*\mathcal{T}_{i_k}^*\ldots\mathcal{T}_{i_1}^* L^e$ from each transfer $\mathcal{T}_{i_k}^*\ldots\mathcal{T}_{i_1}^* L^e$ of length k. The proof that the stochastic iteration scheme converges to the real solution has been proven in [7].

2 Parallel execution of Monte-Carlo algorithms

Monte-Carlo algorithms generate the solution of a problem in the form of a random variable. The mean of this random variable is the real solution and the variance converges to zero with increasing the computational efforts. In this section, we justify that these methods are generally suited for parallel execution. Suppose that two independent versions of a Monte-Carlo algorithm are executed for the solution of the same problem using N samples each, providing random variables ξ_1, ξ_2. The means of these random variables are the same. Generally, the variance of a Monte-Carlo method will be inversely proportional to the number of independent samples used. Thus the variance $\hat{\sigma}^2$ will be the variance σ^2 of a single sample divided by the number of samples, i.e. σ^2/N. Suppose that at the end of the parallel computation, the random variables of the parallel threads are averaged. The mean does not change, however, the variance of the final estimator is:

$$\hat{\sigma}^2 = D^2\left[\frac{\xi_1+\xi_2}{2}\right] =$$

$$\frac{1}{4}\left(D^2[\xi_1] + D^2[\xi_2] + 2\cdot Cov(\xi_1,\xi_2)\right) =$$

$$\frac{\sigma^2}{2N} + \frac{1}{2} Cov(\xi_1, \xi_2)$$

where is $Cov(\xi_1, \xi_2)$ is the covariance of the two random variables. If the two random variables are independent, then the covariance is zero and thus the variance is halved. This independence can usually be provided by inserting a different seed to the random-number generators used by the two processes.

Since the Monte-Carlo estimate is a sum of many independent or weakly correlated samples, it can be supposed to have normal distribution according to the central limit theorem. Examining the shape of the Gaussian probability density, we can conclude that this type of random variables are closer to their mean than three times the standard deviation $\hat{\sigma}$ with probability 0.997. Thus with 99.7% confidence we can say that the probabilistic error bound of the algorithm is $3 \cdot \hat{\sigma}$.

If different parallel implementations need to be compared, the error bound, i.e. the standard deviation is a good measure for comparison. Note that in our case, the algorithm should simultaneously evaluate the 1-bounce, 2-bounce, etc. transfers, and obtain the final result as their sum. The importance of different bounces are not the same, since higher order bounces have less contribution to the final results. Thus it is worth assigning a weight to the number of samples used for the evaluation of different bounces when different algorithms are compared. The contribution of a k order bounce is in average the contribution of a $k-1$ order bounce times the contraction ratio of the integral operator. Let us denote this contraction ratio by a. In global illumination rendering this factor is determined by the average albedo of the scene and by how open the environment is.

Thus if the total contribution of the 1-bounce transfers is C, then the average contribution of the i-bounce transfers is $a^{i-1} \cdot C$. Suppose that a given algorithm generates N_1 samples for the estimation of the 1-bounces, N_2 for the 2-bounces, etc. and N_n samples for the n-bounces while not providing any samples for the bounces of order greater than n. Since the algorithm does not compute the bounces greater than n, the radiance estimate will be biased. The order of this bias error is

$$\epsilon_{\text{bias}} = C \cdot \left(a^{n+1} + a^{n+2} + \ldots\right) = C \cdot \frac{a^{n+1}}{1-a}.$$

Comparing the primary estimators of two different bounces, we can notice two important differences. On the one hand, a higher order bounce has smaller expected value due to the a^{i-1} factor. On the other hand, the estimator of the higher order bounce is a higher dimensional random variable. If we can assume that the second effect is not relevant for the variance of these estimators, then the variance of the i bounce is $a^{2(i-1)} \cdot \sigma_1^2$ where σ_1^2 is the variance of the primary estimator of the first bounce. Assuming that the estimates for the different bounces are independent, the variance of the computed radiance is:

$$\sigma_1^2 \left(\frac{1}{N_1} + \frac{a^2}{N_2} + \frac{a^4}{N_3} + \ldots + \frac{a^{2(n-1)}}{N_n} \right).$$

As stated, the stochastic error bound is 3 times the standard deviation with 99.7% confidence level, thus the probabilistic error bound of the Monte-Carlo estimate is

$$\epsilon_{\text{MC}} = 3\sigma_1 \cdot \sqrt{\frac{1}{N_1} + \frac{a^2}{N_2} + \frac{a^4}{N_3} + \ldots + \frac{a^{2(n-1)}}{N_n}}.$$

The total error of the algorithm is $\epsilon_{\text{bias}} + \epsilon_{\text{MC}}$ in the worst case. Since the total contribution of all bounces is in the order of $C/(1-a)$, the relative error is:

$$a^{n+1} + 3(1-a) \cdot \frac{\sigma_1}{C} \cdot \sqrt{\frac{1}{N_1} + \frac{a^2}{N_2} + \ldots + \frac{a^{2(n-1)}}{N_n}}.$$

When a parallel algorithm is designed, this error must be minimized taking into account the constraint of the computational time and processing power. Parameters such as the contraction a and the relative variance of the first-bounce σ_1/C depend on the scene to be rendered. However, parameters n, $N_1, \ldots, N_n$ are defined by the rendering algorithm, thus they can be controlled to minimize the error.

3 Computational model for parallel stochastic iteration

Suppose that a stochastic iteration scheme is executed on P processors. Each processors iterates I steps independently, then they exchange their results and averaging takes place. For huge scenes, information exchange could mean heavy data transfer. To be general, let us suppose that only Sth portion of the total information is transferred to each processor during an exchange. Setting S to specific values different communication strategies can be modeled. For example, if S is 1, then we get a star topology where each processor can get the results of all other processors. If S is equal to $1/P$, then we can model a round robin scheme, where a processors sends its results only to a single other processors. Finally with arbitrary $S < 1$ value, we can simulate the case when only a fragment of the data is read from each processor in order to reduce the communication overhead.

If there is an exchange after each Ith step, then the total available time T is devoted to N number of iterations and N/I number of exchanges. A single iteration and exchange require T_i and $P \cdot S \cdot T_e$ times, respectively. Thus the total time T is:

$$T = N \cdot T_i + \frac{N \cdot P \cdot S}{I} \cdot T_e, \qquad (1)$$

from which the number of iterations is

$$N = \frac{T \cdot I}{T_i \cdot I + T_e \cdot P \cdot S}. \qquad (2)$$

Crucial design decisions are the appropriate selections of P, S and I, i.e. determining the number of processors that can effectively be used, the fraction of the information exchanged, and after how many iterations the processors should exchange their results. Increasing the number of processors adds more computational power but also increases the communication time, thus we can reach a level where adding new processors does not increase the speed. The frequency and the scale of information exchange are also a matter of contradicting criteria. On the one hand, if I is too small or S is too big, then the frequent information exchange may slow down the process. On the other hand, if I is large or S is small, then the samples produced by the different processors are not combined with each other, which decreases the number of samples. In the next section, this problem is approached as an optimization problem, and the optimal I and S values are determined.

4 Calculation of the sample numbers

Let us first consider an iteration when the processors run independently, and let us denote the number of generated paths of length k at step n by $s_n(k)$. Since in each step each processor introduces a new 1-bounce sample:

$$s_{n+1}(1) = s_n(1) + P \qquad (3)$$

On the other hand a single processor stores every Pth transfer and each processor advances all transfers by one while also keeping the previous samples, thus we can write:

$$s_{n+1}(k+1) = P \cdot \frac{s_n(k)}{P} + s_n(k+1) =$$
$$s_n(k) + s_n(k+1), \qquad k \geq 1. \qquad (4)$$

Now we assume that before executing a single iteration step, the processors exchanged their information. As before, each processor introduces a new 1-bounce sample:

$$s_{n+1}(1) = s_n(1) + P. \qquad (5)$$

On the other hand, a single processor stores now every Sth previous transfer and each processor advances all the stored transfers by one while also keeping the previous samples, thus we can write:

$$s_{n+1}(k+1) = P \cdot S \cdot s_n(k) + s_n(k+1), \qquad k \geq 1. \qquad (6)$$

A complete run of the algorithm consists of

$$K(I) = \frac{N}{I} = \frac{T}{T_i \cdot I + T_e \cdot P \cdot S}.$$

number of phases. Having applied the previous formulae, the stochastic error can be obtained at the end of the algorithm. The error will be a function of P, I and S. The design objective is to minimize the variance as a function of the free parameters.

In order to demonstrate the results, we carried two kinds of numerical experiments. First, we assumed that the available computation time is 1 minute, measured the iteration and exchange time on a Origin 2000 SG computer ($T_i = 4.5$ sec, $T_e = 0.05$ sec) and obtained the number of iterations accordingly. Figures 1 and 2 show the error curves for different number of processors for different length of independent operation.

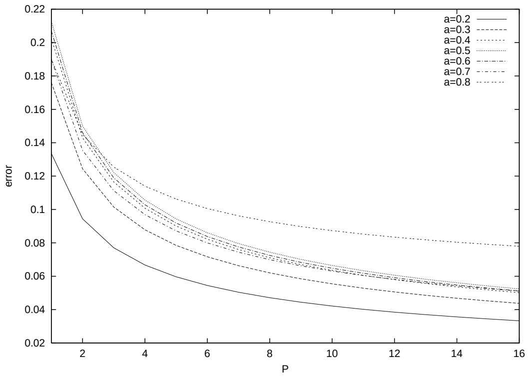


Figure 1: *Stochastic error as a function of the processors P for different contractions ($T = 1$ min, $\sigma_1/C = 1$, $P = 8$, $S = 1$)*

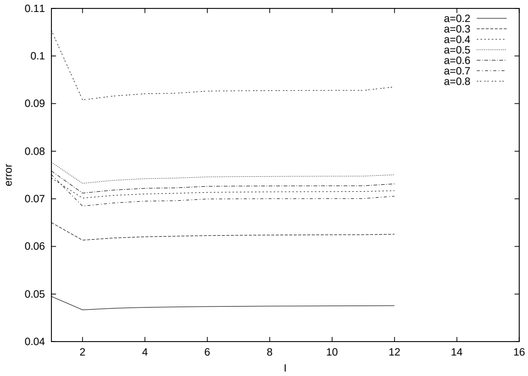


Figure 2: *Stochastic error as a function of the length of the independent iteration cycles for different contractions ($T = 1$ min, $P = 8$, $S = 1$)*

Note that according to the error curves the introduction of a new processor increases the accuracy and the length of independent cycles and the fraction of the exchanged infor-

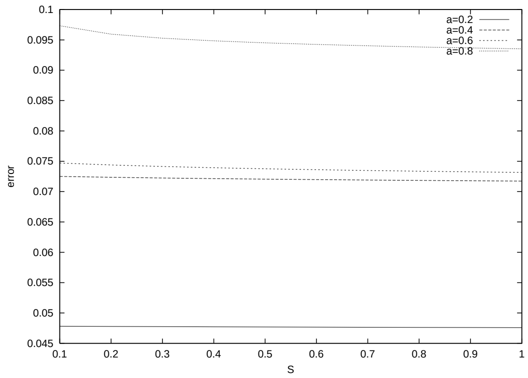


Figure 3: *Stochastic error as a function of the fraction of information exchanged after each step ($T = 1$ min, $P = 8$, $I = 1$)*

mation do not affect the error significantly. This phenomenon can be explained in the following way. Exchanging information increases the samples used for the higher order bounces, which are significant only if the contraction is close to one. On the other hand, frequent and large scale information exchanges steal time from the processors thus they can compute less number of samples, which increases both the Monte-Carlo error and the bias if the contraction is close to one. The two effects seem to well compensate each other.

In the second kind of experiments, the error is fixed to 10% and we examined the computation time required by 1–16 processor systems. The results are in figure 4, which exhibits an interesting feature. If the contraction is high, then the introduction of additional processors only slightly decrease the computation time. This is due to the fact, that higher order bounces need high iteration numbers in which parallelization cannot help.

5 A simplified analytic model: effective sample number

In the previous section an algorithm was presented to find the sample numbers and finally the stochastic error. Unfortunately, the result of this algorithm cannot be obtained in closed form and used directly as an analytical goal function. In this section a simplified

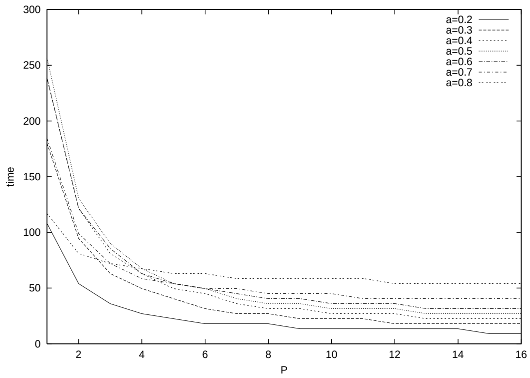


Figure 4: *Computation time as a function of the number of processors*

approach is used that allows analytical treatment. For the sake of notational simplicity, we assume that S is 1. Since the variance is a quadratic operator, we shall assume that the contribution of a k bounce to the general variance of the estimator is a^{2k} times the contribution of the 1-bounces. This leads to the definition of *effective sample number*, which is the weighted number of samples used for the estimation of different bounces:

$$\mathcal{E} = \sum s(k) \cdot a^{2k} \qquad (7)$$

where $s(k)$ is the number of samples generated for the estimation of the k-bounces. When developing parallel algorithms, this effective sample number is intended to be maximized.

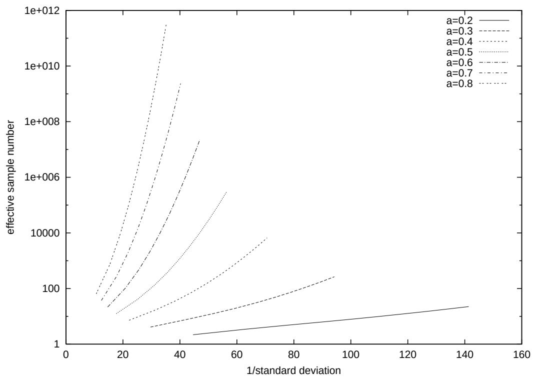


Figure 5: *Effective sample number as a function of the inverse standard deviation*

In figure 5 the effective sample number was plotted against the stochastic error obtained by the formulae of the previous section. Note that there is a monotonous dependence between the two quantities for any practical contraction value, i.e. they can be used instead of the other when ranking different algorithms.

Let us first consider an iteration when the processors run independently. Using equations 3 and 4 a recursive expression can be obtained for the number of effective sample number $\mathcal{E}_{n+1}$:

$$s_{n+1}(1) + s_{n+1}(2)a^2 + \ldots + s_{n+1}(n+1)a^{2(n+1)} =$$
$$(s_n(1) + P) + \ldots + (s_n(n) + s_n(n+1))a^{2(n+1)} =$$
$$P + (a^2+1)\mathcal{E}_n.$$

This recursive expression can be expanded, thus the effective number after $I-1$ steps is

$$\mathcal{E}_{I-1} = P + (a^2+1)\mathcal{E}_{I-2} =$$
$$P + (a^2+1)(P + (a^2+1)\mathcal{E}_{I-3}) =$$
$$P + (a^2+1)P + \ldots (a^2+1)^{(I-2)}P + (a^2+1)^{(I-1)}\mathcal{E}_1 =$$
$$P\frac{(a^2+1)^{(I-1)} - 1}{a^2} + (a^2+1)^{(I-1)}\mathcal{E}_1$$

Assume that before executing a single iteration step, the processors exchanged their information. Substituting equations 5 and 6 into the formula of effective sample numbers we can obtain $\mathcal{E}_{n+1} = P + (Pa^2+1)\mathcal{E}_n$.

Now a complete phase is examined which starts with a combined step then executes $I-1$ independent steps. Merging the previous results together, the effective sample $\mathcal{E}_I$ number at the end of the phase is:

$$P\frac{(a^2+1)^{(I-1)} - 1}{a^2} + (a^2+1)^{(I-1)}(P + (Pa^2+1)\mathcal{E}_1) =$$
$$P\frac{(a^2+1)^{I} - 1}{a^2} + (a^2+1)^{(I-1)}(Pa^2+1)\mathcal{E}_1 = A + B\mathcal{E}_1$$

A complete run of the algorithm consists of $K(I)$ number of phases, thus the effective sample number at the end of the algorithm can be obtained as

$$\mathcal{E}_N = A + B\mathcal{E}_{N-I} = A + B(A + B\mathcal{E}_{N-2I})) =$$
$$A + BA + B^2A + \ldots B^{K-1}A + B^K\mathcal{E}_0 = A\frac{B^K - 1}{B-1}$$

since $\mathcal{E}_0$ is zero. Note that this formula is available in analytic form, thus its derivative can be made equal to zero in order to find the optimum point.

6 Results

To demonstrate the theoretical model a parallel version of a Monte-Carlo radiosity algorithm was considered [4]. In each step this algorithm selects bundles of parallel rays that are parallel with a random direction. The radiosity of the patches that see each other in this direction are exchanged, multiplied with the diffuse reflection and averaged to the patch radiosity. In three different experiments the total numbers of samples, so called bundles, were 50, 100 and 200, respectively. In a P processor system one processor computed $50/P$, $100/P$ and $200/P$ samples. The information of different processors was combined just once at the end of the run. The simulations were executed on a Origin 2000 SGI computer with shared memory. For the 50, 100 and 200 total sample numbers, figure 6 shows the RMS errors against the different independent operation lengths I and the computation times against the number of processors P. Note that according to the theoretical results, the lack of frequent combinations for multi-processor systems caused just a negligible error increase and the algorithm scales well on multi-processor systems. The less the albedo, the less significance the higher order inter-reflections have, thus the need for exchanging information between the processors diminishes. Figure 7 shows two images computed with 1 and 8 processors, respectively. The number of samples was 200. The scene has 1166 polygons that were divided to 19792 patches. The scene in figure 8 has 1130 polygons that were divided to 102804 patches. We used 200 bundles and the total execution time was 204 seconds with 8 processors. The same scene was computed with a single processor in 1282 seconds.

7 Conclusions

This paper presented a theoretical model for the analysis of the efficiency of stochastic iteration global illumination algorithms. The model allowed to study the effect of the algorithm parameters and parallelization strategies on the error of the computation, and also to propose optimal settings for these parameters. Using theoretical considerations and also simulations, we concluded that stochastic iteration can handle the interdependency problem of classical iteration algorithms. It means that with the proper randomization of the algorithm, the threads of different processors can run almost independently and we do not have to slow down them with frequent information exchange.

Acknowledgements

This work has been supported by the OTKA ref. No.: T029135, by TIC 98-0586-C03-02 and the Spanish-Hungarian Fund, ref. No.: E9.

REFERENCES

[1] A. Chalmers and J. Tidmus. *Practical parallel processing, An introduction to problem solving in parallel.* Thomson Computer Press, 1996.

[2] T. Davis, A. Chalmers, and H. Jensen. Practical parallel processing for realistic rendering. *ACM SIGGRAPH*, 2000.

[3] C. Fend and S. Yang. A parallel hierachical radiosity for complex scenes. In *Parallel Rendering Symphosium*, 1997.

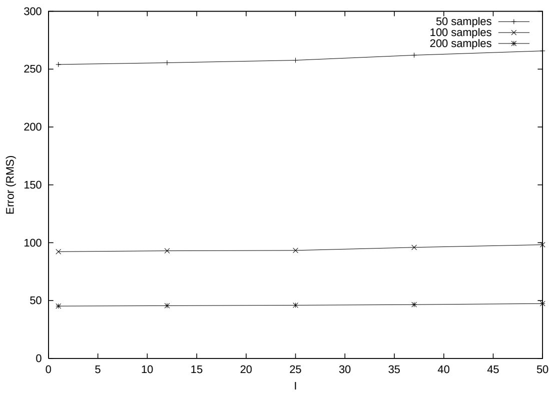
[4] R. Martínez, M. Sbert, and L. Szirmay-Kalos. Adaptive multipath with bundles of parallel lines. In *Proceedings of Visual2000, 3rd International Conference on Visual Computing*, Mexico D.F., 2000.

[5] D. Meneveaux and K. Boutouch. Synchronisation and load balancing for parallel hierachical radiosity of complex scenes on a hegerogeneous computer network. *Computer Graphics Forum*, 18(4):201–212, 1999.

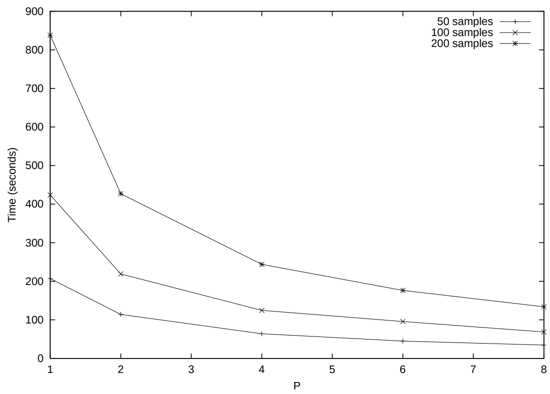
[6] E. Reinhard, A. Chalmers, and F. Jansen. Overview of parallel photo-realistic rendering. *Eurographics 98, State of the Arts Reports*, pages 1–25, 1999.

[7] L. Szirmay-Kalos. Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum (Eurographics'99)*, 18(3):233–244, 1999.

[8] D. Zareski, B. Wade, P. Hubbard, and P. Shirley. Making global monte carlo methods useful: An adaptive approach for radiosity. In *In 1995 Parallel Rendering Symposium, ACM SIGGRAPH*, pages 45–54, 1995. ISBN 0-89791-774-1.



(a)



(b)

Figure 6: *(a) Error versus the length of independent cycles I and (b) computation time versus the number of processors number for 50, 100 and 200 bundles.*

(a)

(b)

Figure 7: *Images obtained with 200 bundles. The total execution time was (a) 838 seconds with 1 processor and (b) 99 seconds with 8 processors*

Figure 8: *Image obtained with multipath with 200 bundles. Radiosities were computed in 204 seconds with 8 processors and in 1282 seconds with a single processor.*