Volume 16 (1997), number 1 pp. 43–55

COMPUTER GRAPHICS *forum*

Parallel Progressive Ray-tracing†

Irena Notkin and Craig Gotsman

Department of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel
E-mail: {cyrus|gotsman}@cs.technion.ac.il

Abstract

A dynamic task allocation algorithm for ray-tracing by progressive refinement on a distributed-memory parallel computer is described. Parallelization of progressive ray-tracing is difficult because of the inherent sequential nature of the sample location generation process, which is optimized (and different) for any given image. We report on experimental results obtained from our implementation of this algorithm on a Meiko parallel computer. The three performance measures of the algorithm, namely, load-balance, speedup, and image quality, are shown to be good.

Keywords: Ray Tracing, Adaptive Sampling, Load Balancing.

1. Introduction

One of the main goals of contemporary computer graphics is efficient rendering of photorealistic images. Optical phenomena must be accurately modeled by the rendering algorithm in order to provide visual realism. Unfortunately, methods rendering accurate images by simulating these physics, such as ray-tracing or radiosity, are computationally very expensive, sometimes requiring minutes of CPU time to produce a medium resolution image of reasonable quality. For this reason the terms "efficient" and "photorealistic" remain conflicting and computer graphics users have to choose between slow high-quality images and fast low-quality images.

This paper is concerned with the ray-tracing method. Much effort has been invested in accelerating this rendering algorithm (see survey in Glassner[1]), but even then, seconds of CPU time are still required to produce an image of reasonable quality and resolution on a high-end workstation. This is obviously unpractical for time-critical applications, such as visual simulation and virtual reality systems, where image sequences are to be generated at almost real-time rates (approximately 20 frames/sec), even if we were willing to compromise somewhat on the resolution and quality of the images.

2. Parallel Ray Tracing

The advent of cheap parallel processing power motivates its use in accelerating ray-tracing, to approach real-time rates. Whitted[2] first observed that ray-tracing lends itself easily to parallelization, as each ray can be traced independently from others by any processor of a parallel computer. Since then many systems have been proposed to exploit this inherent source of parallelism in a variety of ways (see surveys in Green[3] and Jansen and Chalmers[4]). The two main factors influencing the design and performance of parallel ray-tracing systems, are the *computation model* and the *load-balancing mechanism.*

2.1. Computation Models

The two main models of parallel computation, are *demand-driven* computation and *data-driven* computation. In demand-driven systems, each processor is allocated tasks to perform and is responsible for *all* computations related to those tasks. In demand-driven ray-tracing, the task assigned to a processor can be a region of the image space, and that processor is responsible for all computations related to the tracing of all rays spawned by primary rays passing through that region.

† A preliminary version of this paper was presented at the 2nd Winter School on Computer Graphics and Visualization, Pilsen, Czech Republic, February 1995.

All data required by the processor for the computation is communicated to it.

In data-driven systems, processors are allocated different sections of the data, and a computation is assigned to the processor which has access to the data required to perform that computation. Thus, one computational task is performed by a number of processors sharing the required data. In data-driven ray-tracing, processors are allocated parts of the scene geometry and each processor is responsible for all computations accessing the objects contained in its portion of the scene, independent of the origin of the ray being traced. Rays spawned at one processor, but requiring the data of another processor, are transferred to that processor for further handling.

The most important consideration influencing the choice of a particular computation model for ray-tracing is the availability of memory. Because of unpredictable trajectories of secondary rays contributing to the pixel, it is almost impossible to know in advance which objects of the scene will contribute to the tracing of a given primary ray. Therefore, to use a demand-driven model, each processor must have easy access to the entire scene geometry. Not always is there enough local memory to store the entire scene geometry and special techniques are used to perform data exchange, which can seriously degrade performance. Data-driven parallelization uses memory more efficiently. However, as the number of processors increases, the efficiency of the system drops due to substantial task communication overhead.

2.2. Load Balancing

Load balancing mechanisms attempt to guarantee that each processor performs an equal part of the total computation (in terms of CPU time). The general need for a good load balancing technique is amplified by the unpredictable nature of the ray-tracing process, i.e. a large variance in the time required to trace ray trees spawned by different primary rays. It is almost impossible to determine apriori which rays will be "harder" to compute or spawn more rays, and which scene objects will be referenced more often than others, and, as a result, one heavily loaded processor may reduce drastically the performance of the whole system. Figures 5(a) and (f) show images, and Figures 5 (b) and (g) maps of their computational complexity (ray-tracing CPU time). The complexity of a pixel is represented by a proportional gray level intensity. For the image of Figure 5(a), the ratio in complexity between different pixels reaches three orders of magnitude.

There are two main strategies for load balancing, *static* and *dynamic*. The former maps tasks to processors apriori based on *preliminary* estimates of load distribution, so usually cannot ensure good balance. However, since no communication overhead is required while processing, this can sometimes result in an efficient computation. The latter maps tasks to processors on the fly and thus can regulate load distribution at run time, providing better load balancing, at the expense of some monitoring overhead.

3. Progressive Ray Tracing

3.1. Adaptive Sampling

Ray tracing over a regular pixel grid leads to redundant computations on the one hand, and is prone to aliasing artifacts on the other. It has been shown on many occasions (see e.g. Dippe and Wold[5], Mitchell[6] and Mitchell[7]) that nonuniform sampling yields artifacts that are much less noticeable, trading off the aliasing for some noise.

Computer-generated images do not exhibit uniform local image intensity variance. Edges and silhouettes are areas of high contrast, containing high frequencies which require dense sampling, while large, uniform objects and backgrounds are regions with small local variance and do not require high density sampling. However, too sparse sampling of large regions can miss small objects and isolated features. The main goal of any algorithm using nonuniform sampling is to produce a high quality antialiased image with a relatively small number of samples. *Adaptive* nonuniform sampling generates a sample pattern tailored to the image content.

3.2. Progressive Sampling

In most adaptive ray-tracing implementations, *pixels* are supersampled by a varying number of primary rays. Cook[8] used two levels of sampling density, a regular coarse pattern for most pixels and a higher-density pattern for pixels with high variance. Lee et. al[9] varied the sampling density at each pixel continuously as a function of local image variance. In both cases, at least one sample is performed per pixel, and the target image pixel values are computed as the average of the sample values obtained for that pixel. The image is not complete until *all* pixels have been sampled at least once. Painter and Sloan[10] first proposed to treat the image as a continuous region in the plane (without pixel boundaries), adding samples using *progressive refinement*. The advantage of their method is that every prefix of the generated sample set is "optimally" distributed over the image plane, allowing the quick reconstruction of a low quality image from that prefix. Even though only a rough approximation of the final product can be achieved with a small number of samples, it is sometimes very useful to see this rough estimate, which can be further refined if needed. In time-critical applications, the sampling is terminated when time runs out, and some image, possibly crude, is displayed.

At the heart of any progressive ray-tracer lies the adaptive sample generator. The sample generator of Painter and Sloan maintains a binary 2-D tree splitting the two-dimensional image space along x and y axes in turn. The decision on which region to refine by the next sample is based on a variance estimate of the region, its area and the number of samples already in it. In this way the refinement process is driven by two criteria: area coverage and feature location. After regions reach the size of pixels, the only criteria used is mean variance. The refinement process stops when a particular confidence level of the image intensity is reached.

Other[11, 12, 13] sample generators have been proposed for producing an "optimal" sampling pattern. The sample location generator of Eldar et. al [12] (designed for image compression) maintains a growing Delaunay triangulation[14] of sample locations. These triangles are continuously refined. A new sample location is always the center of one of the so-called "Delaunay circles", namely, circles circumscribing the Delaunay triangles. By definition, these circles are empty of other sample points. This next sample location is chosen from the Delaunay circle according to some weighted product of its size and local image intensity variance. In this way it is guaranteed that large regions are refined before smaller ones in order to locate isolated features, and regions containing high frequencies are refined before uniform areas in order to provide anti-aliasing. Figures 5(c) and (h) show sample patterns for the images of Figures 5 (a) and (f) generated by this method. The main data structures needed for the algorithm are the geometric Delaunay triangulation (the latter is also used for image recontruction, see Section 3.3) and a priority queue of 2D points (centers of Delaunay circles). The space complexity of these structures is $O(n)$, where n is the number of sample points. Generating the $(n+1)$'th point involves popping the priority queue, requiring $O(1)$ time, updating the Delaunay triangulation, another $O(1)$ time, and adding new candidate points to the priority queue, another $O(\log n)$ time.. We refer the reader to Eldar et al[12] for further algorithmic details.

3.3. Image Reconstruction

To produce a regular array of image pixels, the irregular color intensity samples are interpolated to the entire plane and resampled at the fixed regular pixel positions. A natural and simple interpolation method is triangulation of the sample set, and *piecewise linear* interpolation on this triangulation. The coordinates (x_p, y_p) of any pixel in the triangle whose vertices are $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ may be expressed as the affine combination $(x_p, y_p) = \alpha(x_1, y_1) + \beta(x_2, y_2) + \gamma(x_3, y_3)$, where α, β, γ are real and non-negative such that $\alpha + \beta + \gamma = 1$. The RGB intensities for that pixel are taken to be $\alpha I_1 + \beta I_2 + \gamma I_3$, where I_i are the intensities of the samples at the triangle vertices. If more than one sample falls within a pixel, the pixel RGB intensities are taken as the average of the samples. Figures 5(d) and (i) show the Delaunay triangulation of the sample sets of Figures 5(c) and (h), and Figures 5(e) and (j) show the piecewise-linear reconstruction of the images based on these triangulations.

4. Parallel Progressive Ray Tracing

4.1. Goals

Adaptive sampling over the continuous image plane speeds up ray-tracing by distributing ray-traced samples in the image areas where they are most needed, thereby reducing the number of rays traced in order to achieve a given image quality. Progressive ray-tracing generates the sample locations in an order such that images may be reconstructed from any prefix of the entire sample pattern. Despite these savings, this method of ray-tracing is still too time-consuming for many applications, as a large number of rays are still required to produce an image of acceptable quality and some overhead is imposed by the sample generation algorithm and image reconstruction.

Suprisingly, the issue of *parallelization* of progressive ray-tracing has not been dealt with in the literature despite its obvious advantage. Standard strategies for parallelization of regular (pixel-based) ray-tracing are not suitable for parallel progressive ray-tracing. The difficulty arising in parallelization of progressive ray-tracing is the inherent sequential nature of the sample location generation algorithm. The location of the ray to be cast next relies heavily on the locations and the values returned from the tracing of all previous rays, implying that processors cannot make independent decisions about where to sample next, but need to see the results of other processors. If care is not exercised, this will result in a suboptimal sampling pattern and hence, suboptimal image quality, relative to that achievable by the serial version of the algorithm.

In the following sections, we describe an algorithm for parallel progressive ray-tracing. Our aim is to design an algorithm suitable for a general-purpose distributed-memory multiprocessing system, based on progressive refinement of the image using up to a fixed number of samples. Besides the standard performance measures to be optimized by any parallel system, speedup and load balancing, our objective is to generate an image of quality approaching that produced by the serial version of the progressive ray-tracing algorithm with the same number of samples.

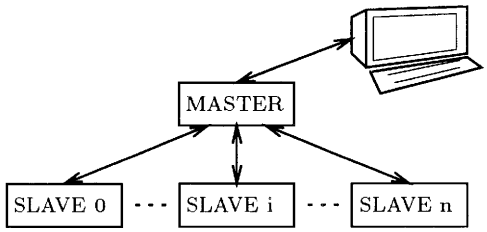

Figure 1: *Parallel system configuration.*

The focus of this work is on the new concerns which arise during parallelization of the *progressive* sampling process. Hence, we ignore standard issues in parallel ray-tracing, such as data-distribution, and assume that the entire scene geometry can be held in the local memory of each processor.

4.2. Computation Model

In the absence of memory limitations, the demand-driven computation model is the most natural to use, since it supports more flexible and efficient load-balancing mechanisms. In our system we have two types of processors: *master* and *slave*. The master monitors the performance, adjusts the work distribution and provides system-user interface, including image display. Slaves run the main progressive ray-tracing tasks. The system configuration is shown in Figure 1. It is very similar to the so-called "processor farm" approach[15].

4.3. Progressive Ray-Tracing Implementation

We use a sample location generator based on that of Eldar et. al[12] (see Section 3.1). The termination criteria is the number of samples allotted to render the image. Run on one processor, the pseudo-code of this algorithm appears in Figure 2.

Samples are evaluated (ray-traced) using the public-domain MTV ray-tracing package[16]. The code supports ray-object intersections with geometric primitives such as spheres, cones and polygons. The bounding volumes acceleration technique is used with an algorithm for ray-volume intersections due to Kay and Kajiya[17]. *Shadow caching* optimization speeds up the search for objects located between the intersection point and the light source.

5. The Parallel Algorithm

5.1. Task Granularity

Our parallel algorithm is based on dynamic task allocation, but has also a static allocation component, determined during the preprocessing stage. The type and granularity of tasks were designed in order to distribute the work as evenly as possible between processors, and in order to achieve a good sample pattern, i.e. as close as possible to that produced by the serial algorithm.

Algorithm **Serial(s,r)**

```
// Adaptive ray-tracer for s samples on
// image region r.

sample_set := build_initial(r);
for i:= 1 to s do
begin
  (x,y) := sample_loc_gen(sample_set);
  c := ray_trace(x,y);
  sample_set := sample_set + {(x,y,c)};
end
return <sample_set,cpu_time,variance>;
```

Figure 2: *The serial adaptive ray-tracing algorithm:* `sample_set` *is a growing set of sample locations and values.* `build_initial` *generates an initial sample set containing 5 samples inside the region* `r`. `sample_loc_gen` *supplies a new sample location based on all previous sample locations and values, and* `ray_trace` *is the ray-trace procedure.* `cpu_time` *is the computation time for sample generation and* `variance` *the variance of the set of sample values.*

Two parameters, namely, an image region specification and the number of samples to be performed in that region, define a task allocated to a processor. A region is the union of square image tiles of fixed size. The regions assigned to each processor are determined during the preprocessing stage, and fixed during the remainder of the algorithm. The number of samples per region is a parameter that may be adjusted according to the progress of the refinement process in the various tiles comprising the region. This provides a chance of achieving good load-balancing, since the distribution of work can then be based on up-to-date information. Hence, we combine static subdivision of image space, aimed at achieving preliminary balanced work distribution, with dynamic distribution of samples between tiles, aiming to optimize the sample pattern.

5.2. Preprocessing

The image plane is divided into a square number of tiles n, a parameter of the algorithm depending on the number of processors p (obviously $n \geq p$). Ultimately, these n tiles are distributed between the p processors, and each processor will thereafter work *only* on the set of tiles assigned to it, which need not be adjacent in the image plane. In order to guarantee a relatively balanced distribution of work between processors, a short preprocessing stage is performed where image characteristics of the tiles, namely, the local variance and image complexity, are estimated. We perform preprocessing in parallel,

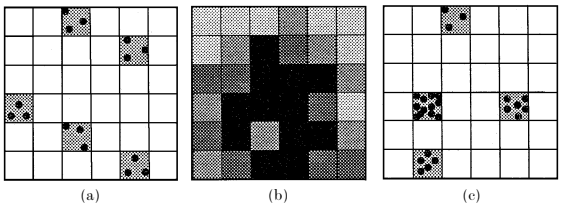

Figure 3: *Algorithm for 8 slaves and 36 tiles.* **(a)** *Preprocessing work of slave 2 in tiles i such that* $i \pmod 8 = 2$*).* **(b)** *Tile weight map built by master after preprocessing stage. Gray level intensity is proportional to weight.* **(c)** *Processing by slave 2 in tiles assigned to it by master according to results of (b). Note that this set of tiles is different from that of (a).*

where each slave processor is temporarily assigned an (almost) equal number of tiles, and performs a small number *pr* of adaptive samples in it (see Figure 3). No attempt at balancing the load of these minute tasks is made. The cpu time required for the samples in each tile, cpu_time(t), is measured, as is their variance, var(t).

The results of the preprocessing stage are sent to the master processor who then assigns weights $w(t)$ to each tile t:

$$w(t) = \text{cpu\_time}(t) \cdot pr(t) \tag{1}$$

where

$$pr(t) = \max_{T \in t} \{ \text{rad}(T) \cdot \log(1 + \text{var}(T)) \} \tag{2}$$

The maximum is taken over all triangles T in the Delaunay triangulation of the sample locations in tile t, rad(T) is the radius of the circle circumscribing the triangle T, and var(T) is the variance of the three intensities obtained at the vertices of T. The logarithmic function on the variance was adopted from Eldar et al, who propose a number of different weighting functions. This one seemed to yield the best sample patterns.

The master then assigns (afresh) tiles to slave processors in a way that the tile weight is more or less uniformly distributed between the processors. It is obvious that such crude preliminary estimates will not be sufficient to provide good load balancing, so this tile distribution serves only as a base for future on-line (dynamic) load balancing.

5.3. On-Line Load Balancing

We distribute the responsibility of achieving the goals of our algorithm between master and slaves in the following manner: the master provides balanced load distribution while slaves distribute well the samples (whose number is assigned by the master) in their tiles. The main parallel adaptive-ray-tracing procedure consists of the master assigning tasks to p slaves on demand until the total required number of samples for the image, s, is reached. The basic task, assigned by the master processor to a slave processor, is ray-tracing of some number of samples (in its region). The initial task size is a parameter $k \leq s/p$, but is decreased with time, in order to prevent the case where the slave processor assigned the last task works alone on a large number of samples, after the others have already finished. The parameter $0 < d \leq 1$ determines the rate of decay of the task size.

The slave processors work *exclusively* on the set of tiles assigned to them at the preprocessing stage. A separate sample location data structure is maintained by the slave processor for each of its tiles. On receipt of a task, the slave processor distributes the alloted samples between its tiles, which are ordered in a priority queue. The priority $pr(t)$ of a given tile t is calculated as in (2), so large unsampled areas with large variance have the highest priority and thus are sampled first.

The slave processor pops the highest priority tile from the queue and performs in it a small number of samples, using the serial progressive ray tracing algorithm (Figure 2). We call these smaller tasks *mini-tasks*. The size of the mini-task performed by the slave processors is a parameter q. After a mini-task has been performed on a tile, its priority is updated and it is inserted back into the priority queue. Obviously, if the priority of some tile is much larger than the priorities of all other tiles in the queue, many mini-tasks will be performed in that tile before another tile is treated. The mini-tasks are performed until the number of samples specified by the master processor for the task is exhausted.

Reconstruction of the image is done by the master processor after the completion of sampling by all slave processors. The master collects *all* sample locations and values from the slave processors and (serially) reconstructs the entire image by the method described in Section 3.3.

Pseudo-code of our algorithm appears in Figure 4.

6. Experimental Results

6.1. The Parallel Architecture

Our algorithm was implemented on a Meiko general-purpose parallel computer with distributed memory. It consists of a SparcStation1 host processor with 28MB RAM and 28 i860 processors with at least 8MB RAM each. The topology of the system is logically reconfigurable. The peak performance of the i860 processor is 120 MIPS.

Algorithm **Dynamic(p,s,**n,k,d,q,pr)

```
// Adaptive ray-trace s samples by p processors.
// Use n image tiles. Initial task size is k
// samples, which decreases with decay rate d.
// Mini-task size is q. pr samples per tile are
// performed during preprocessing.

MASTER:
   // preprocess by receiving results from slaves.
   for t:= 1 to n do
      <cpu[t],var[t]> := receive(t%p);

   map tiles to slaves according to weights based
     on <cpu,var>;          // see Eq. (1)

   // initialize slave task sizes.
   for proc := 0 to p-1 do k[proc] := k;

   // main ART loop.
   s -= n*pr;
   while s>0 do
     for proc := 0 to p-1 do
       if idle(proc) and s>0 then
       begin
         assign_task(proc,k[proc]);  // tell slave proc
         s -= k[proc];               // to perform task
         k[proc] *= d;       // of size k[proc] samples
       end;

   terminate all slave processors;    // terminate ART.

   // collect results from slaves.
   samples = {};
   for proc := 0 to p-1 do
      samples = samples U receive(proc);

   // reconstruct image from samples.
   reconstruct(samples);

SLAVE:
   // preprocess pr samples per tile.
   for t := 1 to n do
      if (t%p==slave_id) then
      begin
         <samp,c,v> := serial(pr,tile[t]);
         send(<c,v>) to master;
      end;

   // main ART procedure.
   samples = {};
   while (not terminated) do
   begin
     k := request_task();  // receive task from master.
     while k>0 do
       t := pop(priority_queue);  // get tile.
       <samp,c,v> := serial(q,tile[t]);
       samples = samples U samp;
       prior := update(tile[t].priority); // see Eq. (2)
       push(priority_queue,t,prior);
       k -= q;
     end;
   end;
   send(samples) to master;        // send results.
```

Figure 4: *The parallel progressive ray-tracing algorithm.*

6.2. Performance Measures

The performance of our parallel progressive ray-tracing algorithm for p processors and s samples is evaluated using the following measures :

1. **Speedup**

$$S(p,s) = \frac{T(1,s)}{T_{max}(p,s)}$$

$T(1,s)$ is the CPU time required for one processor to perform the serial progressive ray-tracing algorithm with s samples and $T_{max}(p,s)$ is the CPU time consumed by the "slowest" processor of the p-processor parallel system, running the parallel progressive ray-tracing algorithm with s samples. The times do not include the reconstruction of the image, which is performed serially by the master processor only. Ideally, $S(p,s) = p$.

2. **Load disbalance**

$$L(p,s) = \frac{T_{max}(p,s) - T_{min}(p,s)}{T_{min}(p,s)}$$

$T_{max}(p,s)$ and $T_{min}(p,s)$ are the CPU times consumed by the "slowest" and the "fastest" of p processors, while tracing s rays in total, respectively. Ideally $L(p,s) = 0$.

3. **Image infidelity** (unique for ART)

$$N(p,s) = ||I(p,s) - I(1,s)||_1$$

$I(p,s)$ and $I(1,s)$ are the pixel color intensities of the image produced by the parallel algorithm and the serial algorithm, respectively. $||\cdot||_1$ is the mean l_1 norm:

$$||(r,g,b)||_1 = \frac{1}{3n}\sum_{i=1}^{n}(|r_i| + |g_i| + |b_i|) \quad .$$

Ideally $N(p,s) = 0$.

6.3. Test Scenes

We tested our algorithm on scenes obtained[18] from the Standard Procedural Database (SPD) of Haines[19]. The SPD was designed to be a standard benchmark for evaluating the performance of rendering algorithms. We present here the results for the SPD models "gears" (Figure 5(a)) and "tree" (Figure 5(f)) (**See page 49 for Figure 5**). "Gears" is a polygonal scene, containing a variety of complex silhouettes and multiple edges, which challenge adaptive sampling algorithms. Apart from this, the "gears" model contains objects with various material properties, giving rise to intensive reflection and refraction processes. As a result, the CPU time required to trace different primary rays varies within three orders of magnitude (see Figure 5(b)), a fact which significantly affects task distribution. "Tree" contains many very small features (e.g. leaves), which could easily be lost in a progressive ray-tracing scenario.

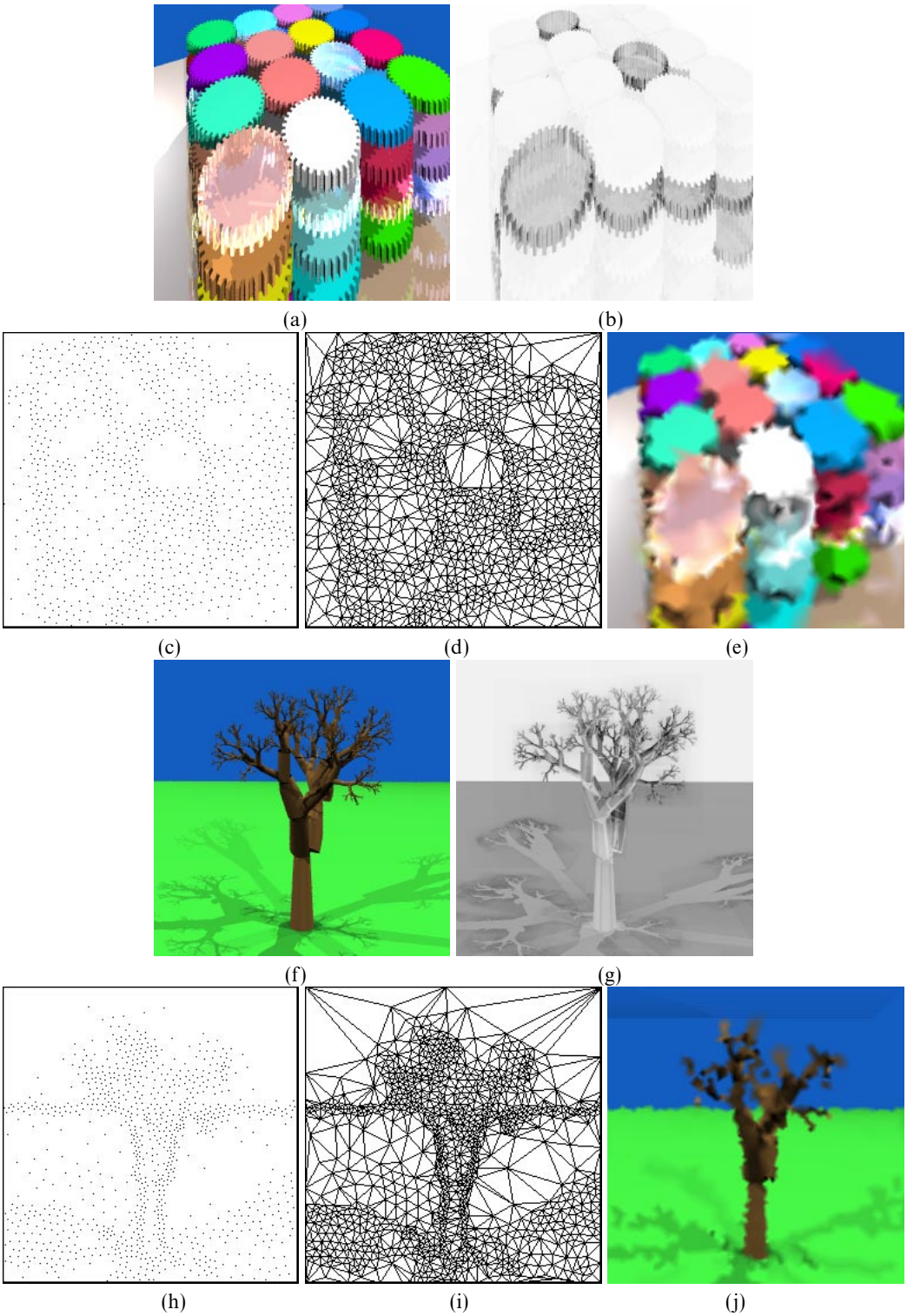

Figure 5: (a) *Synthetic image "gears".* **(b)** *Complexity map. Darker colors denote pixels requiring more CPU time to compute* **(c)** *Adaptive sample pattern of 1,000 primary rays, concentrated mostly in areas of high image intensity variance. Note that there is no correlation between ray complexity and image intensity variance.* **(d)** *Delaunay triangulation of the sample pattern of (c).* **(e)** *Piecewise linear image reconstruction based on the triangulation (d).* **(f)-(j)** *Analogous to (a)-(e) for "tree" scene.*

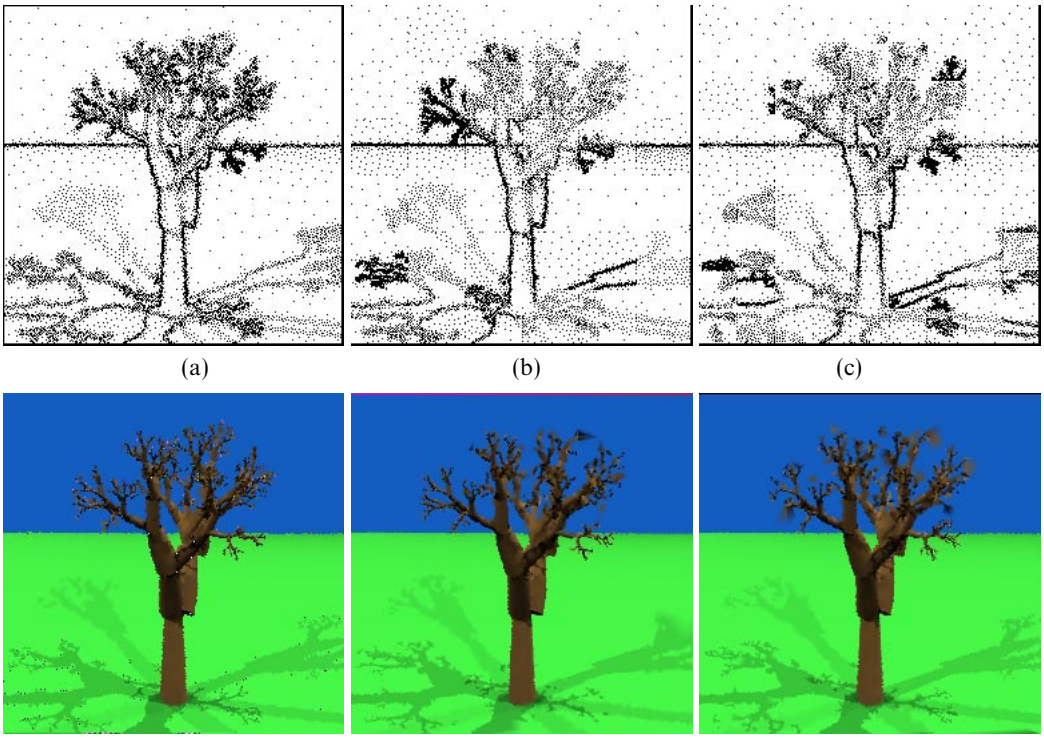

Figure 7: *Qualitative performance of the parallel adaptive ray-tracing algorithm with* $s = 10,000$ *primary rays on the "tree" scene: sample patterns and reconstructed images produced by various numbers of processors:* **(a),(d)** $p = 1$ *(serial).* **(b),(e)** $p = 14$. **(c),(f)** $p = 26$. *Note the reconstruction artifacts in the branch shadows in the extreme right portion of (e).*

6.4. Quantitative and Qualitative Results

The sample patterns and reconstructed images produced by our parallel algorithm, sampling 10,000 primary rays in total, by various numbers of processors, are shown in Figures 6 and 7 (**See page 52 for Figure 6**). Obviously, the number of tiles grows with the number of processors in order to provide well-distributed task allocation. A large number of tiles (81 in the case of 26 processors) influences the quality of the sample patterns and, hence, also the reconstructed images. This is reflected in so-called *edge effects* which occur because processors do not distribute samples well in the vicinity of tile borders. On the other hand, reducing the number of tiles may lead to significant disbalance in work distribution which, in turn, damages sample patterns. The patterns shown in Figure 6 demonstrate the tradeoff between these two factors. Even so, certain badly-sampled regions (waste or lack of samples as compared to the serial pattern) in the "worst" pattern (produced by 26 processors) lead to artifacts in the reconstruction which are hardly noticable.

Figure 8 shows quantitative results of our algorithm for the test scenes. The speedup and disbalance (Figure 8(a-b,d-e)) are very good. For 26 processors working on the "gears" scene, we achieve speedup of 23.8 and load disbalance of 15%. The image reconstruction results in an image infidelity of 11 (on a scale of 0-255) for "gears" and 5 for "tree", which is very good, considering the relatively small number of samples performed.

Attention should be paid to the fact that when the performance of the rendering algorithm is measured, the most important measure is still the qualitative one, i.e. visual appreciation of the image (and its artifacts). In our case, we consider our results to be quite good.

6.5. Algorithm Parameter Values

We initially obtained the best values for the algorithm parameters n, k, d, q and pr, which depend on the scene, and on p and s, by trial and error. After some practice, we obtained rules of thumb for determining these values. The best number of image tiles, n, is determined by the following tradeoff: A large number of tiles enables each processor to work on a variety of tiles located in different regions of the image, so balances the load better. Too many tiles damages the sample pattern significantly, as "edge effects", related to samples on the tile borders, dominate them. The values we used for n

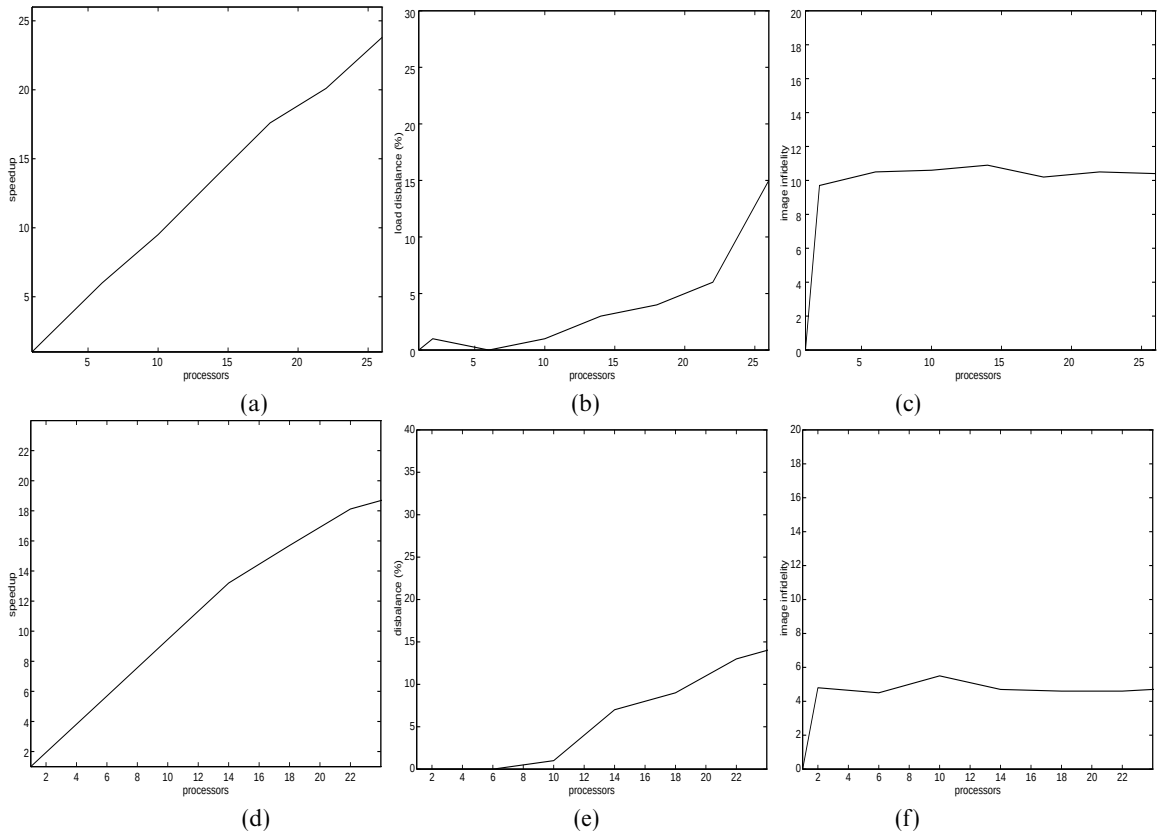

Figure 8: *Quantitative performance of the parallel adaptive ray-tracing algorithm on the test scenes.* **(a)** *"Gears" speedup.* **(b)** *"Gears" load disbalance.* **(c)** *"Gears" image infidelity.* **(d)-(f)** *Analogous to (a)-(c) for "tree".*

were square integers, for programming convenience. The (simple) rule of thumb is that more tiles are required for larger numbers of processors. The performance does not seem to be too sensitive to this number, as long as it is not too small or too large.

The performance of the algorithm is quite sensitive to the task granularity determined by the initial task size k, and its decay rate d. Increasing task granularity will increase communication overhead. In terms of load balancing, fine granularity is important towards the end of the sampling process, so k and d must be chosen so that the resulting geometric series of task sizes starts off with a relatively large fraction of the samples per processor, and reaches a size of about 2% of the samples per processor at the end. The rule of thumb is that $k \approx s/2p$, so that approximately half the samples to be performed by a processor are assigned already at its first task. The decay rate for the task size, d, should be approximately 30%.

Our algorithm does not seem to be very sensitive to the value of q - the size of the mini-tasks performed by a processor on any one of its tiles. In practice, we took $q = 1$.

We used values between 5 and 10 for pr - the number of preprocessed samples per tile. More samples would provide more reliable estimates of tile weights, but would slow down the algorithm, as no load-balancing is performed during this stage.

Table 1 shows the best values of the n, k and d parameters for our algorithm run on the "gears" scene with $s = 10,000$ and $p = 10$, and how a change in each one of these parameters influences the performance measures S, L and N. Figure 9 shows sample patterns corresponding to each one of parameter vectors of Table 1. Obviously, the resulting suboptimal sample patterns cause a decrease in the quality of the reconstructed images.

7. Conclusion

7.1. Discussion

A fundamental assumption which we made about the underlying architecture is the existence of enough local

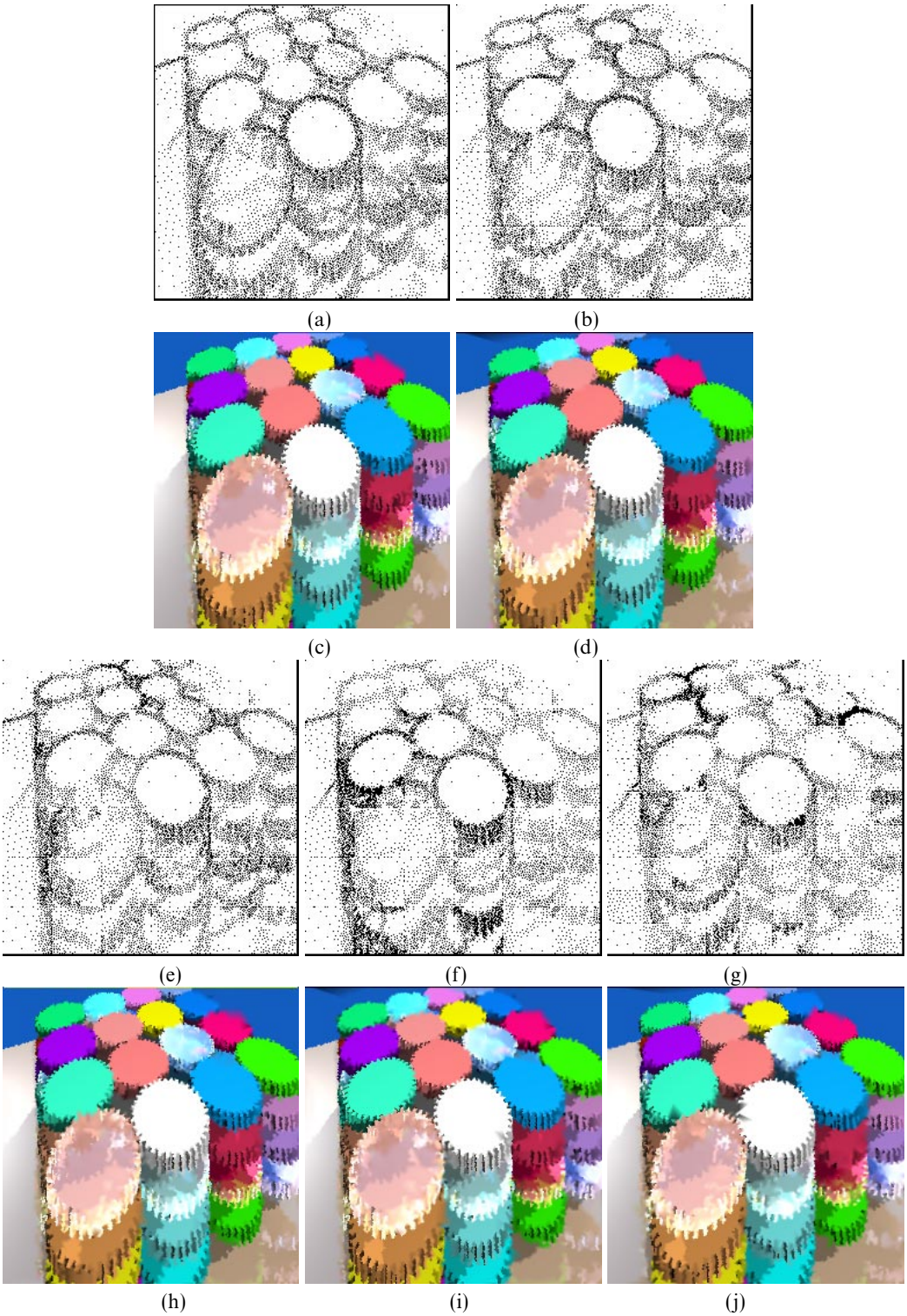

Figure 6: *Qualitative performance of the parallel adaptive ray-tracing algorithm with* $s = 10,000$ *primary rays on the "gears" scene: sample patterns and reconstructed images produced by various numbers of processors:* **(a),(c)** $p = 1$ *(serial).* **(b),(d)** $p = 2$. **(e),(h)** $p = 10$. **(f),(i)** $p = 18$. **(g),(j)** $p = 26$. *Note the reconstruction artifacts in the white gear at the top of the stack just behind the nearest (pink) gear in (i) and (j).*

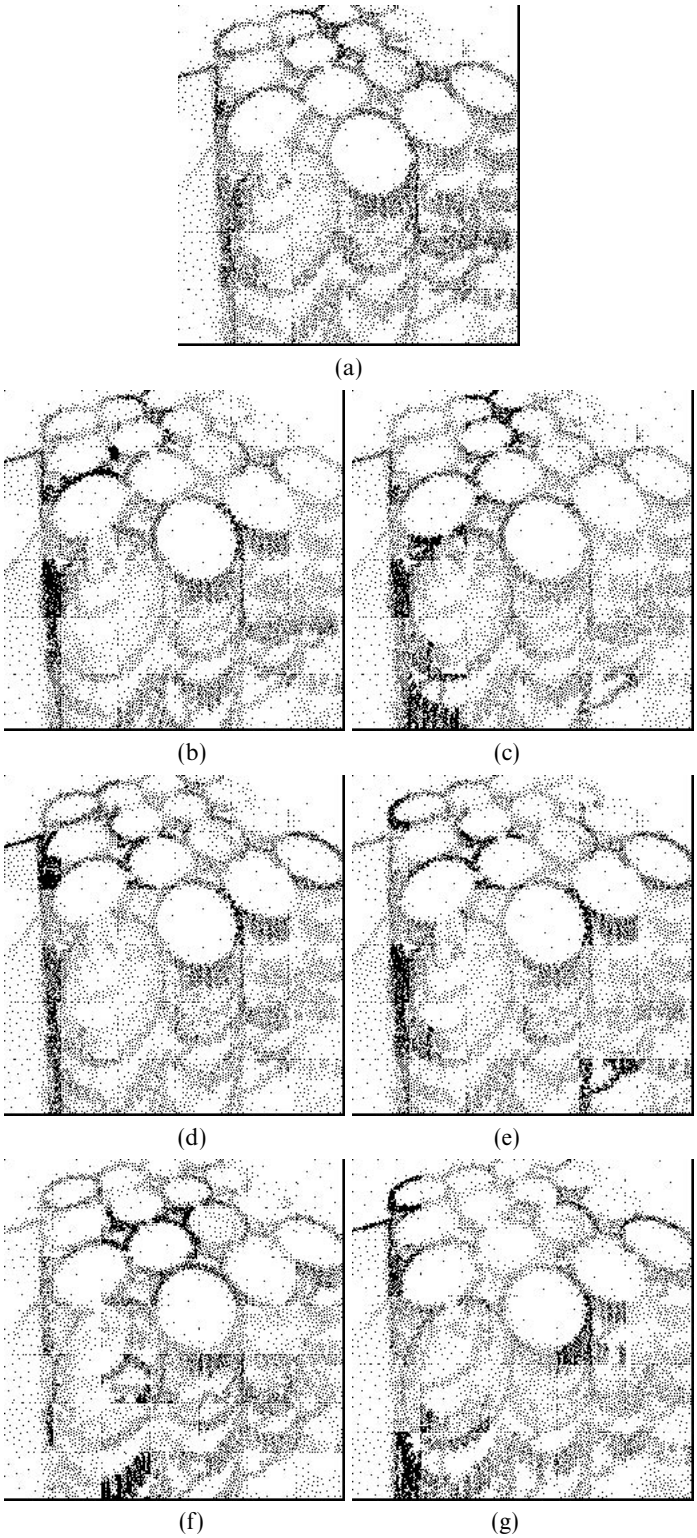

Figure 9: *"Gears" sample patterns resulting from varying one of the algorithm parameters.* $s = 10,000$, $p = 10$. **(a)** *best* ($n = 36, k = 450, d = 30, q = 1, pr = 10$), **(b)** $k = 460$, **(c)** $k = 440$, **(d)** $d = 25$, **(e)** $d = 35$, **(f)** $n = 49$, **(g)** $n = 25$.

n	k	d	S	L	N
36	**450**	**30**	9.9	1	10.1
36	**460**	30	9.1	5	10.3
36	**440**	30	9.1	4	10.1
36	450	**25**	9.6	2	10.1
36	450	**35**	9.5	4	10.3
49	450	30	9.2	4	10.3
25	450	30	9.7	21	10.1

Table 1: *Influence of n, k and d parameters on S, L and N performance measures for "gears", $s = 10,000$, $p = 10$. The first row shows the best values of n, k, d. In each of the other rows, one of the parameters (in boldface) is modified (increased or decreased) causing a degradation in performance.*

processor memory to store the entire scene geometry. The Meiko local processor memory (8-32MB) allowed us to store large scenes, such as "gears", in each processor, but this may not be the case for other parallel machines. In case of memory limitations, a special memory management mechanism must be incorporated into the parallel algorithm.

As was mentioned in Section 6.4, "edge effects" damage sample patterns because processors generate bad sample patterns in their tile border areas. An optimization which probably could diminish this unpleasant effect would require additional communication between processors, aimed at joint sampling of adjacent regions. We did not implement this, as the decrease in reconstructed image quality caused by "edge effects" is negligible compared with the decrease in speedup and load balance, which would be incurred by the additional inter-processor communication within the Meiko platform. Another possible solution to the edge-effects problem is to use overlapping image tiles. The reconstruction algorithm, merging all samples, would get a better density, and suppress some of the edge artifacts. This method incurs an extra sampling cost, and will be checked in future implementations.

7.2. Summary

An algorithm for parallel progressive ray-tracing based on dynamic task allocation has been proposed. The algorithm for parallelization of progressive ray-tracing differs fundamentally from those used for parallelization of standard (pixel based) ray-tracing, since it optimizes the number and locations of the rays traced, which are generated by a process which has an inherent serial nature. Our algorithm significantly (almost to the best possible) speeds up the progressive ray-tracing, which is still too time-consuming to run serially.

Our algorithm has been shown to achieve very good speedup and load-balancing. The price paid for this is a somewhat suboptimal sample pattern, relative to that produced by the serial algorithm. This, however, still results in an image almost identical to that obtained by the serial algorithm, and even to that obtained by a supersampling pixel-based ray-tracing algorithm.

The algorithm is designed for a general-purpose parallel computer with distributed memory and can be easily ported to any other similar architecture with minor modifications.

Performing progressive ray-tracing in parallel may be viewed in a wider context as a special case of parallel progressive sampling of a real function over a continuous domain. This is an important open problem in parallel processing. We have reported first results in that direction.

Acknowledgements

The first author thanks Johann Makowsky for his help during the early stages of this work. Thanks also to Ohad Ben-Shachar for contributing to the implementation of the serial sampling algorithm.

References

1. A. Glassner (Ed.). *An Introduction to Ray-Tracing*. Academic Press, (1989).
2. T. Whitted, "An Improved Illumination Model For Shaded Display", *Communications of the ACM*, **23**(6):343–349, (June 1980).
3. S. Green, *Parallel Processing for Computer Graphics*, Pitman, London, (1991).
4. F. W. Jansen and A. Chalmers, "Realism in Real Time ?" In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pp. 27–46, Eurographics, (1993).
5. M. Dippe and E. Wold, "Antialiasing Through Stochastic Sampling", *Computer Graphics*, **19**(3):69–78, (July 1985).
6. D. P. Mitchell, "Generating Antialiased Images at Low Sampling Densities", *Computer Graphics*, **21**(4):65–69, (July 1987).
7. D. P. Mitchell, "Spectrally Optimal Sampling for Distribution Ray Tracing", *Computer Graphics*, **25**(4):157–164, (July 1991).
8. R. L. Cook, "Stochastic Sampling in Computer Graphics", *ACM Transactions on Graphics*, **5**(1):51–72, (January 1986).

9. M. E. Lee, R. A. Redner and S. P. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing", *Computer Graphics*, **19**(3):61–65, (July 1985).

10. J. Painter and K. Sloan, "Antialiased Ray Tracing by Adaptive Progressive Refinement", *Computer Graphics*, **23**(3):281–287, (July 1989).

11. T. Akimoto, K. Mase and Y. Suenaga, "Pixel-selected ray tracing", *IEEE Computer Graphics and Applications*, **11**(4):14–22, (1991).

12. Y. Eldar, M. Lindenbaum, M. Porat and Y. Zeevi, "The farthest-point strategy for progressive image sampling", In *Proceedings of the 12th International Conference on Pattern Recognition, Jerusalem*, (1994).

13. J.-L. Maillot, L. Carraro and B. Peroche, "Progressive Ray-Tracing", In *3rd Eurographics Workshop on Rendering*, Bristol, 1992. Consolidation Express.

14. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, (1985).

15. D. May and R. Sheperd, "Communicating process computers", Technical Report #22, Inmos Ltd., Bristol, (1987).

16. M. T. Van de Wettering, "MTV's ray tracer, 1989". Available from markv@cs.uoregon.edu.

17. T. Kay and J. Kajiya, "Ray tracing complex scenes", *Computer Graphics*, **20**(4):269–278, (August 1986).

18. E. Haines, "Standard procedural databases", May 1988. Available through Netlib from netlib@anl-mcs.arpa.

19. E. Haines, "A proposal for standard graphics environments", *IEEE Computer Graphics and Applications*, **7**(11):3–5, (November 1987).