Parallel Adaptive Ray-Tracing

Irena Notkin Craig Gotsman

Department of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel

{cyrus|gotsman}@cs.technion.ac.il

Abstract

We describe a dynamic task allocation algorithm for ray-tracing by adaptive progressive refinement on a parallel computer. Parallelization of adaptive ray-tracing is difficult because of the inherent sequential nature of the sample location generation process, which is optimized (and different) for any given scene. 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.

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 intensive, 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.

1.1 Parallel Ray Tracing

The advent of cheap parallel processing power motivates its use in the speedup of many time-consuming computer graphics algorithms. Whitted [14] 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 [4, 7]). The two main factors influencing the design and performance of a parallel ray-tracing system are the model of computation and the *load-balancing* mechanism.

The two main models of parallel computation, described in the literature, are *demand-driven* computation and *data-driven* computation. In demand-driven systems, each processor is allocated certain 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 ray tracing of pixels contained in that region.

In data-driven systems, processors are allocated different sections of the data, and each computation is assigned to the processor which has access to the appropriate data. 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 database and each processor is responsible for all computations accessing the objects contained in its part of the database, 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.

Load balancing mechanisms attempt to guarantee that each processor performs an equal part of the computation. 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 tracing time for different 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. Fig. 5(a-b) shows an image and a map of its computational complexity (ray-tracing cpu time). The complexity of a pixel is represented by a proportional gray level intensity. For the image in Fig. 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 optimal balance. However, since no communication overhead is required while processing, this can sometimes provide 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.

1.2 Adaptive Ray Tracing

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 [2, 10, 11] 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.

Adaptive sampling adjusts the sample rate on the fly to concentrate samples where they are needed most. In the early days of ray-tracing, *pixels* were supersampled by a varying number of primary rays. Cook [1] used two levels of sampling density, a regular coarse pattern for most pixels and a higher-density pattern for pixels of high contrast. 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 was performed per pixel, and the target image pixel values were computed as the average of the sample values obtained for that pixel. Painter and Sloan [12] first proposed to treat the image as a continuous region of the plane (without pixel boundaries), using *adaptive progressive refinement* by recursive subdivision of this region. The advantage of their method is that every prefix of an ordered sample set is "optimally" distributed over the image plane, allowing the quick creation of a low resolution image. 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. Their sample generator 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: 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.

Various other sample generators have been

proposed for producing an "optimal" sampling pattern. The sample location generator of Eldar et. al [3] maintains a growing Delaunay triangulation of sample locations. These triangles are continuously refined. A new sample location is a vertex of the dual Voronoi diagram, which is equally distant from some other sample locations. This next sample location is determined by the triangle maximizing some weighted product of its geometric properties 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. Fig 5(c) shows an adaptive sampling pattern for the image of Fig. 5(a).

To produce a regular array of image pixels, the irregular color function samples are interpolated to the entire plane and resampled at the fixed pixel positions. A possible interpolation method is triangulation of the sample set, and *piecewise linear* interpolation on this triangulation. Fig. 5(d-e) shows the Delaunay triangulation of the sample sets in Fig. 5(c), and piecewise-linear reconstruction of the image based on this triangulation.

2 Parallel Adaptive Ray Tracing

2.1 Goals

Adaptive sampling over the continuous image plane speeds up ray-tracing by optimally distributing ray traced samples in the image areas where they are most needed, thereby reducing the number of rays traced. However, the algorithm still remains 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 adaptive ray-tracing has not been dealt with in the literature despite its obvious advantage. Standard strategies for parallelization of regular ray-tracing are not suitable for parallel adaptive ray-tracing. The difficulty arising in parallelization of adaptive 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 can not make independent decisions about where to sample next without seeing the results of other processors. If care is not exercised, this will result in a suboptimal sampling pattern and hence, suboptimal picture quality.

In this paper we describe an algorithm for parallel adaptive ray-tracing. Our aim is to design an algorithm suitable for a general-purpose multiprocessing system, based on progressive refinement of the image using 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 adaptive ray-tracing algorithm with the same number of samples.

2.2 Computation Model

In this work we assume that the entire scene database can be held in the local memory of each processor. 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 adaptive ray-tracing tasks. The system configuration is shown in Fig. 1.

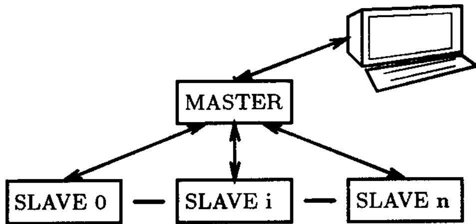

Figure 1: Parallel system configuration.

```
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.

2.3 Adaptive Ray-Tracing Implementation

Our adaptive sampling algorithm uses the progressive refinement principle proposed by Painter and Sloan [12]. We use a sample location generator based on that of Eldar et. al [3] (see Section 1.2). The termination criteria is the number of samples allotted to render the image. Run on one processor, the pseudo-code of this algorithm is described in Fig. 2.

Samples are evaluated (ray-traced) by means of the public-domain MTV ray-tracing package [13]. The code supports ray-object intersections with geometric primitives such as spheres, cylinders, cones and polygons. The bounding volumes acceleration technique is used with an algorithm for ray-volume intersections due to Kay and Kajiya [8]. *Shadow caching* optimization speeds up the search for objects located between the intersection point and the light source.

3 The Parallel Algorithm

Our parallel algorithm is based on dynamic task allocation. 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 an optimal sample pattern, i.e. close to that produced by the serial algorithm. Dynamic task allocation provides the best chance of achieving good load-balancing, since the distribution of work can be done on the fly, based on up-to-date information. The basic task, assigned by the master processor to a slave processor, is serial adaptive ray-tracing of a given number of samples in a given region of the image plane. Such a region is the union of a number of rectangular areas in the image, called *tiles*. These tiles need not be adjacent in the image plane. The slave processor receiving this task distributes the allotted samples between the tiles, a few at a time. We call these smaller tasks *mini-tasks*. The total number of image tiles used by our algorithm is a parameter, n. Obviously $n \geq p$. Once assigned (at the beginning of the process) an image region consisting of a number of tiles, the slave processor maintains a separate sample location data structure for each tile. The slave processor works *exclusively* on these tiles from now on.

In order to determine the mapping of image tiles to processors, a short preprocessing step is performed in parallel, where each slave processor is temporarily assigned an (almost) equal number of tiles, and performs a small number pr of adaptive samples in it. 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 *weight* of a tile t is then defined as:

$$w(t) = \frac{\text{cpu_time}(t)}{\log(\text{var}(t))} \quad (1)$$

The master processor then maps tiles to slave processors such that the tile weight is more or less uniformly distributed between the processors. The main parallel adaptive-ray-tracing procedure then consists of the master assigning tasks to slaves on demand. The slave performs mini-tasks on its tiles, which are ordered in a priority queue.

The priority of a given tile t is calculated as follows:

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

where the maximum is taken over all triangles in the Delaunay triangulation of the sample locations of the 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 reason we use rad(T) (as opposed to, e.g. area(T)) is that the new sample location, associated with the triangle prescribed by the method, is a vertex of the dual Voronoi diagram, which is precisely the center of the circumscribing circle. This way, large unsampled areas with large variance are sampled first. The master processor allocates tasks to slave processors on demand, until the total required number of samples for the image, s, is reached. 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 $0 \leq d \leq 1$ determines the rate of decay of the task size.

The size of the mini-task performed by the slave processors is a parameter q. These mini-tasks are performed until the number of samples specified by the master processor in the task is exhausted. Pseudo-code of our algorithm appears in Fig. 3 and a schematic diagram in Fig. 4. Reconstruction of the image is done by the master process after the completion of sampling

```
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.
   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 3: The parallel adaptive ray-tracing algorithm.

by all slave processes. This is done by piecewise linear interpolation of the intensity values at the vertices over the triangles of the Delaunay triangulation of the *entire* sample set.

4 Experimental Results

4.1 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 28 MB RAM and 28 i860 processors with at least 8 MB RAM each. The logical connection between processors can be reconfigured such that each processor can be connected logically with 8 other processors (through a transputer accompanying the i860). The message-passing system is based on Ethernet communications. The Peak Performance of the i860 processor is 120 MIPS.

4.2 Performance Measures

The performance of a parallel adaptive ray-tracing algorithm 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 adaptive 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 adaptive ray-tracing algorithm with s samples. 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 $L = 0$.

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

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

   $I(p,s)$ is the image produced by the parallel algorithm tracing s rays on p processors. $||\cdot||_1$ is the l_1 norm. Ideally $N = 0$.

4.3 Test Scenes

We tested our algorithm on scenes obtained [6] from the Standard Procedural Database (SPD) of Haines [5]. The SPD was designed to be a standard benchmark for evaluating the performance of rendering algorithms. We present here the results for the SPD model "gears" (see Fig. 5(a)). "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 computational time required to trace different primary rays varies within three orders of magnitude (see Fig. 5(b)), a fact which significantly affects task distribution.

Figs. 6 and 7 show the qualitative and quantitative results of our parallel algorithm with up to 26 processors sampling 10,000 primary rays in total. For the "gears" scene, speedup is close to optimal. The sample patterns generated by our algorithm are quite reasonable, compared to the one generated by the serial algorithm. For 26 processors, we achieve speedup of 23.8. The load disbalance is 15%, indicating that the parameters of the algorithm (n, k, d, q and pr) are fine tuned almost to their optimum. Any other loss in speedup is due to the communication overhead of the parallelization. The image reconstruction is extremely good, resulting in an image infidelity of 11 (on a scale of 0-255).

4.4 Algorithm Parameter Values

We obtained the optimal 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. The optimal number of image tiles, n, is determined by the following tradeoff: A large number of tiles enable the processors to work on a variety of tiles located in different regions of the image, so balances the load better. However, 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 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.

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 did 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 samples per tile during the preprocessing stage. 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.

The optimal values of the parameters for $s = 10,000$ and select values of p are indicated in Fig. 6.

5 Conclusion

Performing adaptive ray-tracing in parallel may be viewed in a wider context as a special case of parallel adaptive 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.

Our parallel adaptive ray-tracing algorithm based on dynamic task allocation has been shown to achieve good speedup and load-balancing. The price paid for this is a somewhat suboptimal sampling pattern (relative to that produced by a serial algorithm), which, however, still results in an image almost identical to that obtained by the serial algorithm.

Acknowledgements

The first author thanks Johann Makowsky for his help during early stages of this work.

References

[1] R.L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[2] M. Dippe and E. Wold. Antialiasing through stochastic sampling. *Computer Graphics*, 19(3):69–78, July 85.

[3] 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.

[4] S. Green. *Parallel Processing for Computer Graphics*. Pitman, London, 1991.

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

[6] E. Haines. Standard procedural databases, May 1988. Available through Netlib from netlib@anl-mcs.arpa.

[7] F.W. Jansen and A. Chalmers. Realism in real time ? In *Proceedings of the Fourth Eurographics Workshop on Rendering*, pages 27–46. Eurographics, 1993.

[8] T. Kay and J. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 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] D.P. Mitchell. Generating antialiased images at low sampling densities. *Computer Graphics*, 21(4):65–69, July 1987.

[11] D.P. Mitchell. Spectrally optimal sampling for distribution ray tracing. *Computer Graphics*, 25(4):157–164, July 1991.

[12] J. Painter and K. Sloan. Antialiased ray tracing by adaptive progressive refinement. *Computer Graphics*, 23(3):281–287, July 1989.

[13] M.T. Van deWettering. MTV's ray tracer, 1989. Available from markv@cs.uoregon.edu.

[14] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

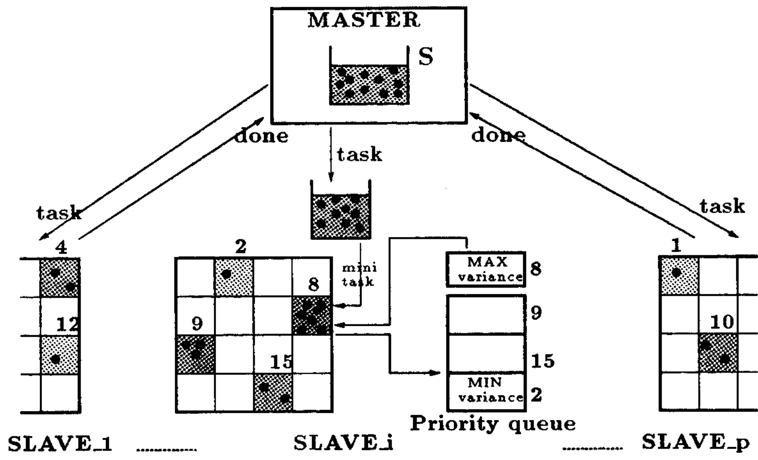

Figure 4: Schematic illustration of the parallel adaptive ray-tracing algorithm.

(a) (b) (c)

(d) (e)

Figure 5: **(a)** Synthetic image "gears". **(b)** Complexity map. Darker regions require more cpu time. **(c)** Adaptive sampling patterns of 1000 primary rays, mostly concentrated 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 sampling patterns of (c). **(e)** Piecewise linear reconstruction based on the triangulation (d).

(a) (b) (c)

(d) (e) (f)

Figure 6: Qualitative performance of the parallel adaptive ray-tracing algorithm on the "gears" scene: Sample patterns and reconstructed images produced by the parallel algorithm with s = 10,000 primary rays for: **(a),(d)** $p = 1$ processor. **(b),(e)** $p = 14$ processors, $n = 36$, $k = 350$, $d = .30$, $q = 1$, $pr = 10$. **(c),(f)** $p = 26$ processors, $n = 81$, $k = 120$, $d = .25$, $q = 1$, $pr = 5$.

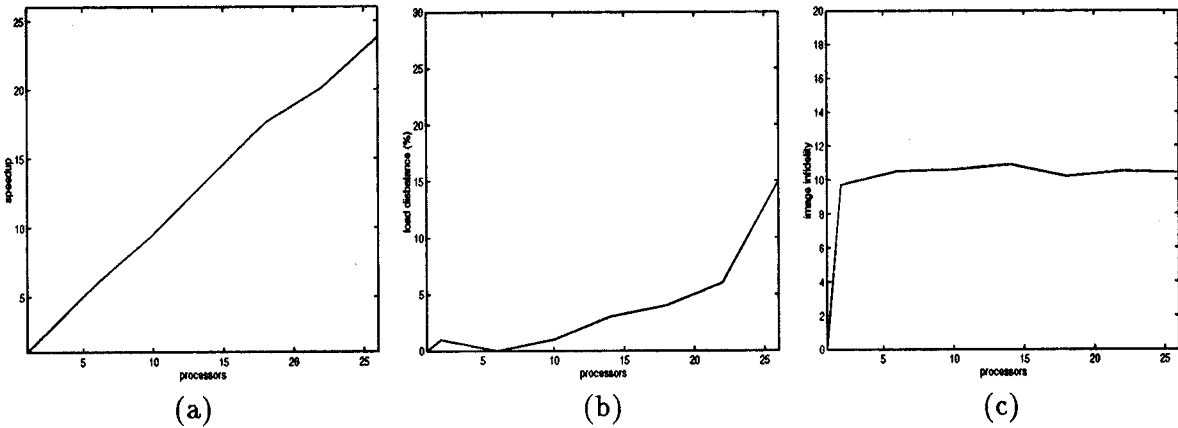

Figure 7: Quantitative performance of the parallel adaptive ray-tracing algorithm on the "gears" scene. (a) Speedup. (b) Load disbalance. (c) Image infidelity.