Alcom-FT Technical Report Series
ALCOMFT-TR-02-95

Perfect Load Balancing (Demand-Driven) Parallel Ray Tracing*

Tomas Plachetka

University of Paderborn, Department of Computer Science, Fürstenallee 11,
D-33095 Paderborn, Germany

Abstract. A demand-driven parallelization of the ray tracing algorithm is presented. Correctness and optimality of a perfect load balancing algorithm for image space subdivision are proved and its exact message complexity is given. An integration of antialiasing into the load balancing algorithm is proposed. A distributed object database allows rendering of complex scenes which cannot be stored in the memory of a single processor. Each processor maintains a permanent subset of the object database as well as a cache for a temporary storage of other objects. A use of object bounding boxes and bounding hierarchy reduces the number of requests for missing data to a necessary minimum. The proposed parallelization is simple and robust. It should be easy to implement with any sequential ray tracer and any message-passing system. Our implementation is based on POV-Ray and PVM.

1 Introduction

Ray tracing [22] computes an image of a 3D scene by recursively tracing rays from the eye through the pixels of a virtual screen into the scene, summing the light path contributions to pixels' colors. In spite of various optimization techniques [5], typical sequential computation times range from minutes to hours.

Parallel ray tracing algorithms (assuming a message passing communication model) can be roughly divided into two classes [6]:

Image space subdivision (or screen space subdivision) algorithms [7], [6], [1], [9], [13], [15] exploit the fact that the primary rays sent from the eye through the pixels of the virtual screen are independent of each other. Tracing of primary rays can run in parallel without a communication between processors. The problem of an unequal workload in processors must be considered. Another problem arises by rendering large scenes—a straightforward parallelization requires a copy of the whole 3D scene to be stored in memories of all processors. On the other hand, these algorithms are usually easy to implement.

Object space subdivision algorithms [4], [3], [17], [2], [10], [21], [8], [14], [1], [9], [20] geometrically divide the 3D scene into disjunct regions which are distributed in processors' memories. The computation begins with shooting of primary rays into processors storing the regions through which the primary rays pass first. The rays are then recursively traced in the processors. If a ray leaves a processor's region, it is passed to the processor which stores the adjacent region (or discarded if there is no adjacent region in the ray's direction). An advantage of object space subdivision algorithms is that the maximum size of the rendered 3D scene is theoretically unlimited because it depends only of the total memory of all processors. Potential problems are an unequal workload and a heavy communication between processors. Moreover, an implementation of these algorithms may be laborious.

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and VEGA 1/7155/20.

Some parallelizations (*hybrid algorithms*) [21], [11], [12], [18] extend the data-driven approach of object space subdivision algorithms with additional demand-driven tasks running in processors in order to achieve a better load balance.

A lot of work has been done concerning ray tracing parallelization. However, the majority of the research papers is interested only in the performance of the algorithms. Software engineering issues are seldom mentioned. Under these belong questions like:

- Can the parallel algorithm be easily integrated into an existing sequential code?
- Will it be possible to continue the development of the sequential code without having to always reimplement the parallel version?

This paper presents a parallelization based on the algorithm of Green and Paddon [7]. The basic idea is to combine the image space subdivision with a distributed object database. Our implementation named POV||Ray (read "parallel POV-Ray") builds upon the (sequential) freeware ray tracer POV-Ray version 3.1. Many practical aspects addressed in the implementation apply generally.

Screen space subdivision using a process farming is explained in section 2. Correctness and optimality of a perfect load balancing algorithm as well as its exact message complexity are proved. It is also shown how antialiasing can be integrated into the load balancing algorithm. Section 3 describes the management of distributed object data: initial object distribution, memory organization, cache and object exchange mechanism. Results of experiments are presented in section 4, followed by conclusions in section 5.

2 Image space subdivision

The computed image consists of a finite number of pixels. Computations on pixels are independent of each other. However, computation times on different pixels differ and they are not known beforehand. A dynamic load balancing is therefore needed for an efficient parallelization. Otherwise some of the processors may be idle at the end of the computation while other processors may continue working for a long time. There are two principal approaches to achieving a balanced load in processors: work stealing [1] and process farming. A problem of both is tuning their parameters. We shall concentrate on process farming.

2.1 Process farming

Process farming uses a central process, a *load balancer*, for distributing image parts to *worker processes* (*workers*) on demand. When a worker becomes idle, it sends a *work request* to the load balancer. Once a worker gets a work (an image part), it must compute it in a whole. Workers do not communicate with each other.

The problem is to determine an appropriate granularity of the assigned parts. The choice of granularity influences the load balance and the number of exchanged messages. There are two extreme cases:

1. The assigned parts are minimal (pixels). In this case the load is balanced perfectly but the number of work requests is large (equal to the number of pixels which is usually much greater than the number of workers).
2. The assigned parts are maximal (the whole image is partitioned into as many parts as the number of workers). In this case the number of work requests is low but the load imbalance may be great.

A compromise between these two extremes is to divide the image in more equal parts than the number of workers. Even better is to assign large parts of the image to idle workers at the beginning and small ones later. This is what the following algorithm does (a version of this algorithm is given in [13] and [15]). The algorithm is perfect in the sense that it guarantees a perfect load balancing, and at the same time it minimizes the number of work requests if the ratio of computation times on any two parts of the same size can be bounded by a constant T. An example of a image subdivision generated by the algorithm is shown in Figure 1.

Perfect load balancing algorithm

```
float T;    /* a constant such that time(part1)/time(part2) <= T, T >= 1.0
               for any two parts part1, part2 of the same size */
int W;      /* the total number of atomic parts
               (e.g., image pixels or image columns) */
int N;      /* the number of workers
               (running on a homogeneous parallel machine) */
{
  int job_size;
  int work = W;
  while (work > 0)
  {
    part_size = max(1, floor(work/(1 + T * (N - 1))));
    for (counter = 0; counter < N; counter++)
    {
      ... wait for a work request from an idle worker;
      if (work > 0)
      {
        ... send work of size part_size to the idle worker;
        work = work - part_size;
      }
    }
  }
  ... collect work requests from all workers;
  ... send termination messages to all workers;
}
```

Claim. The previous algorithm always assigns as much work as possible to idle workers, while still ensuring the best possible load balance.

Proof. The algorithm works in rounds, one round being one execution of the while-loop. In the first round the algorithm assigns image parts of size

$$s_{max} = max\,(1, \lfloor \mathrm{W}/(1 + T \cdot (\mathrm{N}-1)) \rfloor)$$

(measured in the number of atomic parts). In each of the following rounds the parts are smaller than in the previous round. Obviously, the greatest imbalance is obtained when a processor p_{max} computes a part of the size s_{max} from the first round as long as possible (whereby the case of $s_{max} = 1$ is trivial and will not be considered here) and all the remaining $N-1$ processors compute the rest of the image as quickly as possible (in other words, the load of the remaining $N-1$ processors is perfectly balanced). The number of parts computed in parallel by all processors except of p_{max} is $W - \lfloor \mathrm{W}/(1 + T \cdot (\mathrm{N}-1)) \rfloor$. The ratio of the workload (in terms of the number of atomic parts) of one of the $N-1$ processors (let us denote it p_{other}) and s_{max} is then

$$\frac{\frac{W - \lfloor \mathrm{W}/(1+T\cdot(\mathrm{N}-1)) \rfloor}{N-1}}{\lfloor \mathrm{W}/(1+T\cdot(\mathrm{N}-1)) \rfloor}$$

This ratio is greater or equal to T. This means that the processor p_{other} does at least T times more work than the processor p_{max} in this scenario. From this and from our assumptions about T and about the homogeneity of processors follows that the processor p_{max} must finish computing its part from the first round at the latest when p_{other} finishes its part from the last round. Thence, a perfect load balance is achieved even in the worst case scenario.

It follows directly from the previous reasoning that the part sizes s_{max} assigned in the first round cannot be increased without affecting the perfect load balance. (For part sizes assigned in the following rounds a similar reasoning can be used, with a reduced image size.) This proves the optimality of the above algorithm. □

Claim. The number of work requests (including final work requests that are not going to be fulfilled) in the previous algorithm is equal to

$$N \cdot (r+1) + \left\lceil W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^r \right\rceil$$

where

$$r = \max\left(0, \left\lfloor \log_{1+\frac{N}{1+T\cdot(N-1)}} (W/N) \right\rfloor\right)$$

Proof. It is easy to observe that

$$\frac{N \cdot W}{1 + T \cdot (N-1)} \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^{i-1}$$

atomic parts get assigned to workers during the i^{th} execution of the while-loop and that

$$W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^i$$

atomic parts remain unassigned after the i^{th} execution of the while-loop.

r is the total number of executions of the while-loop minus 1. The round r is the last round on the beginning of which the number of yet unassigned atomic parts is greater than the number of workers N (note that $r = 0$ if the number of atomic parts would be less than or equal to N after the first execution of the while-loop). r can be determined from the fact that the number of yet unassigned atomic parts after r executions of the while-loop is less than N:

$$W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^r \leq N$$

which yields (r is an integer greater than 0)

$$r = \max\left(0, \left\lfloor \log_{1+\frac{N}{1+T\cdot(N-1)}} (W/N) \right\rfloor\right)$$

There are N work requests received during each of the r executions of the while-loop, yielding a total of $N \cdot r$ work requests. These do not include the work requests received during the last execution of the while-loop. The number of work requests received during the last execution of the while-loop is equal to

$$\left\lceil W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^r \right\rceil$$

Finally, each of the workers sends one work request which cannot be satisfied. Summed up,

$$N \cdot (r+1) + \left\lceil W \cdot \left(1 - \frac{N}{1 + T \cdot (N-1)}\right)^r \right\rceil$$

is the total number of work requests. □

Two parameters must be tuned in the algorithm. The first parameter is the constant T. The second parameter is the size of an atomic work part in pixels (it makes sense to pack more pixels into a single message because a computation of several pixels usually costs much less than sending several messages instead of one). We experimentally found (see section 4) that a combination of T between 2.5 and 4.0 and an atomic part size of a single image column yields the best performance for most scenes (if no antialiasing is applied). Figure 2 shows the total number of work requests as a function of image resolution for a varying number of workers.

Fig. 1. Assignment of work parts in the perfect load balancing algorithm for two workers and $T = 3.0$

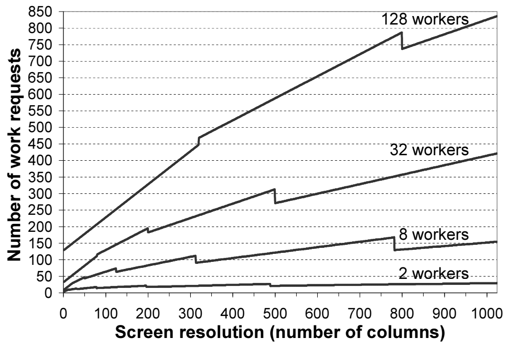

Fig. 2. The number of work requests in the perfect load balancing algorithm. Atomic parts are image columns, $T = 3.0$, image width varies from 0 to 1024

We use a process farm consisting of three process types in our implementation. The *master* process is responsible for interaction with the user, for initiation of the ray tracing computation, and for assembling the image from the computed image parts. The *load balancer* process runs the perfect load balancing algorithm, accepting and replying work requests from workers. *Worker* processes receive parts of the image from the load balancer, perform (sequential) ray tracing computations on those parts and send the computed results to the master process. The main idea of this organization is to separate the process of image partitioning (which must respond as quickly as possible to work requests in order to minimize the idle periods in workers) from collecting and processing of results.

A special attention by an implementation of the process farm should be devoted to the scene parsing phase (a distribution of a large scene onto workers may be very time consuming when a shared file system is used on the parallel machine running the worker processes). The problem may be overcome by parsing the scene by chunks in the master process and broadcasting the chunks to workers. [15]

Furthermore, the scene should persist in the worker processes, especially by computations of ray-traced walk-through animations of large scenes. Instead of having to reparse a large scene file, only a short camera file must be then incrementally parsed after each animation frame. [15]

Another important problem—an efficient implementation of antialiasing—is discussed in the following section.

2.2 Antialiasing

It is useful to restrict the work parts to rectangular areas because they are easy to encode (a rectangular work part is identified by coordinates of its upper-left and bottom-right corners). If we further restrict the work parts to *whole neighboring columns* or *whole neighboring rows*, then columns should be used by wider-than-taller images and rows otherwise. This leads to a finer granularity of image parts in the perfect load balancing algorithm (because of the integer arithmetic used in the algorithm).

A good reason for using *whole* image columns or rows as atomic parts is an integration of antialiasing in the load balancing algorithm. A usual (sequential) antialiasing technique computes the image using one ray per pixel and then for each pixel compares its color to the colors of the left and above neighbors. If a significant difference is reported, say, in the colors of the pixel and its left neighbor, both pixels are resampled using more primary rays than one (whereby the already computed sample may be reused). Each pixel is resampled at most once—if a color difference in two pixels is found, whereby one of them has already been resampled, then only the other one is going to be resampled. Particular pixels by this antialiasing strategy are the pixels of the left-most column and upper-most row, respectively. In order to compare the colors of the pixels in the left-most column to the colors of their left neighbors, an additional column on the left side of the image must be computed (which does not belong to the image). Similarly, an additional line must be computed on the upper side of the image. The computation of these additional two lines is not a great problem by a sequential algorithm. However, in a parallel algorithm using an image subdivision the additional columns and lines must be computed for each image part generated by the image subdivision because the pixels to the left or above may be computed by some other processor. This leads to a duplication of work. An alternative to work duplication is an additional communication. The additional work or communication is minimized when whole image columns (or rows) are used for the image partitioning.

We solve the described problem using additional communication. No work is duplicated in workers. If antialiasing is required by the user, the workers apply antialiasing to their image parts except of *critical parts*. (A critical part is a part requiring information that is not available on that processor in order to apply antialiasing.) Critical parts are single image columns (from now on we shall assume that whole image columns are used as atomic image parts). A worker marks pixels of a critical part which have already been resampled in an *antialiasing map* (a binary map). Once a worker finishes its computation of an image part, it packs the aliasing map and the corresponding part of the computed image to the job request and sends the message to the load balancer. The load balancer runs a slightly modified version of the perfect load balancing algorithm from section 2.1. Instead of sending the termination messages at the end of the original algorithm, it replies idle workers' requests with *antialiasing jobs*. An antialiasing job consists of a critical part and the part of which is the critical part dependent. The computed colors and antialiasing maps of these two parts (two columns) are comprised in the antialiasing job. Only after all critical parts have been computed, the load balancer answers all pending work requests with termination messages. A worker, upon a completion of an antialiasing job, sends as before another job request to the load balancer and the resulting image to the master process. The master process updates the image in its frame buffer.

The computation of antialiasing jobs is interleaved with the computation of "regular" jobs. There is no delay caused by adding the antialiasing phase to the original load balancing algorithm. The antialiasing phase may eventually *compensate* a load imbalance caused by an underestimation of the constant T in the load balancer.

Note that the above reasoning about using whole columns or rows as atomic image parts does not apply to antialiasing jobs. The image parts of antialiasing jobs can be horizontally cut into rectangular pieces to provide an appropriate granularity for load balancing.

3 Distributed object database

A disadvantage of image subdivision is that the parallel algorithm is not data-scalable, that means, the entire scene must fit into memory of each processor. This problem may be overcome by using a distributed object database. We followed the main ideas described for instance in [7], [6], [18], [19]. In the following we talk about our implementation.

Before coming to a design of a distributed object database, let us do some observations. A 3D scene consists of objects. Ray tracers support a variety of object types most of which are very small in memory. Polygon meshes are one of a few exceptions. They require a lot of memory and occur very frequently in 3D models. If a scene does not fit into memory of a single processor, it is usually because it consists of several objects modeled of large polygon meshes. However, any single object (or any few objects) can usually be stored in memory. Moreover, it should also be said that a vast majority of scenes *does* fit into memory of a single processor.

For the above reasons we decided to distribute polygon meshes in processors' memories. All other data structures are replicated—in particular object bounding hierarchy including the bounding boxes of polygon meshes. At the beginning of the computation data of each polygon mesh reside in exactly processor (we shall refer to the processor as to the *mesh's owner*). Each processor must be able to store—besides the meshes it owns and besides its own memory structures—at least the largest polygon mesh of the remaining ones.

When speaking about memory constraints on a processor, we mean the processor's real memory. Each worker is assumed to have a fixed amount of real memory to its disposal in every moment. Actually, in concurrent systems (in Unix, for instance) it is very difficult to assign a fixed amount of real memory to a certain process—it is up to the user to not allow other memory-consuming processes to run on the processors running the worker processes.

The initial distribution of objects on processors is quasi random. The master process is the first process which parses the scene. When it recognizes a polygon mesh, it makes a decision which of the worker processes will be the object's owner. The master process keeps a track of objects' ownership and it always selects the worker with the minimum current memory load for the object's owner. Then it broadcasts the mesh together with the owner's ID to all workers. Only the selected owner keeps the mesh data (vertices, normals, etc.) in memory. The remaining processes preprocess the mesh (update their bounding object hierarchies) and then release mesh's data. However, they do not release the *object envelope* of the mesh. In case of meshes, this object envelope contains the mesh's bounding box, an internal bounding hierarchy tree, the size of the missing data, the owner's ID, a flag whether the mesh data are currently present in the memory, etc.

A mesh's owner acts as a server for all other processors who need those mesh's data. When it receives a mesh request from some other processor (meshes have their automatically generated internal IDs known to all processors), it responds with the mesh's data. Two things are done at the owner's side in order to increase efficiency. First, the mesh's data are prepacked (to a ready-to-send message form) even though it costs additional memory. Second, the owner runs a thread which reacts to mesh requests by sending the missing data without interrupting the main thread performing ray tracing computations. (This increases efficiency on parallel computers consisting of multiprocessor machines connected in a network, such as Fujitsu-Siemens hpcLine used for our experiments.)

The mesh data are needed at two places in the (sequential) ray tracing algorithm: by intersection computations and by shading. The problem of missing object data is solved by inserting an additional code at these two places. This code first checks whether the mesh's data are in memory (this information is stored in the mesh's envelope). If the data are not present, the processor checks whether there is enough memory to receive and unpack the data. If there is not enough memory, some objects are released from the *object cache memory* (we use Last Recently Used replacement policy which always releases objects that have not been used for the longest time from the cache to make space for the new object) and then sends a data request to the owner. Upon the arrival of the response, the data are unpacked from the message into the cache memory. After that the original computation is resumed.

Efficiency can be slightly improved also on the receiver's side. Note that in the ray tracing algorithm shading of a point P is always preceded by an intersection test resulting in the point P. However, there can be a large number of intersection and shading operations between these two. A bad situation happens when the mesh data needed for shading of the point P have meanwhile been released from the cache. In order to avoid this, we precompute all necessary information needed for shading of point P in the intersection code and store it in the mesh's object envelope. The shading routine is modified so that it looks only into the mesh's envelope instead of into mesh's data. By doing so we avoid an eventual expensive communication for the price of a much less expensive unnecessary computation (not all intersection points are going to be shaded).

There is a communication overhead at the beginning of the computation also when the cache memory is large enough to store the entire scene. Recall that it must be assumed during the parsing phase that the entire scene will not fit into processors' cache memories, and therefore mesh data are released in all processors but one after they have been preprocessed. Cache memories of all processors are thus empty after the parsing phase. This is called a *cold start*. A cold start can be avoided by letting the user decide whether to delete meshes from processor's cache memories during parsing. If an interaction with the user is not desirable and if computed scenes may exceed a single processor's memory, then a cold start cannot be avoided.

Note that the above mechanisms can be applied to any object types, not only to polygon meshes. Certain scenes may include large textures which consume much memory. It makes then sense to distribute textures over processors' memories as well. The cache replacement policy (LRU, for instance) can easily be extended to handle any object types.

4 Experiments

The following results were measured on the hpcLine parallel computer by Fujitsu-Siemens. hpcLine consists of double-processor nodes (Pentium III, 850 MHz) with 512 MB of memory per node. PVM 3.4 uses Fast Ethernet network for communication.

We used a model of a family house for the measurements. It consists of about 614 objects (approximately $75,000$ triangles) and 22 light sources. We used the PAL resolution (720x576 pixels) in the renderings.

Speedup and efficiency graphs are shown in Figure 3. The use of distributed database was switched off in these measurements. $T = 4.0$ (see section 2.1) was used for all measurements without antialiasing. In the measurements with antialiasing, 16 samples per pixel were used when the colour difference of neighboring pixels exceeded 10%, and the constant T was set to 16.

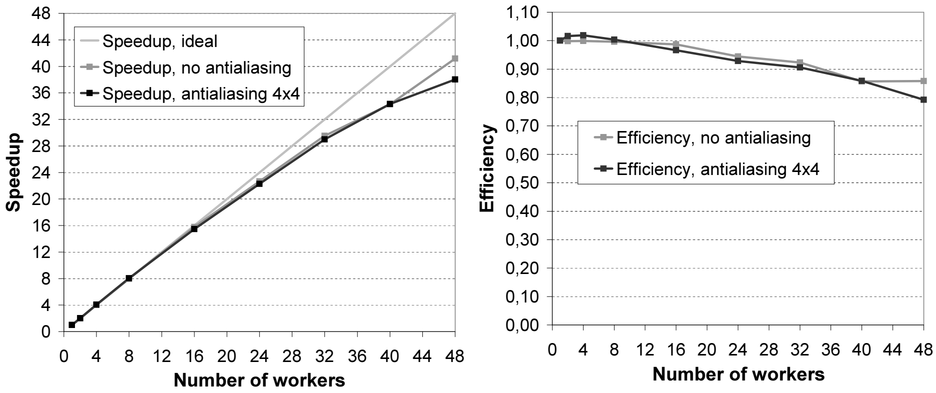

Fig. 3. Speedup and efficiency measurements (no distributed database used)

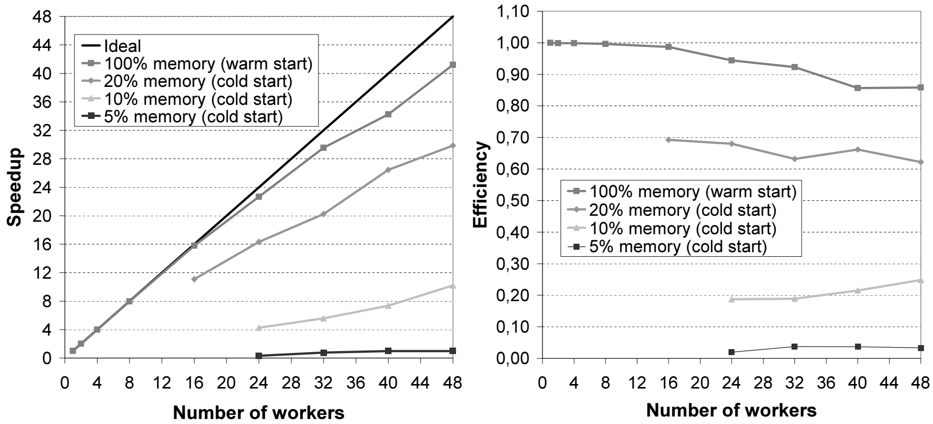

Fig. 4. Speedup and efficiency measurements (distributed database used)

The memory limit was simulated in the software by the experiments with the distributed object database (the whole scene can be stored in the memory of one node of hpcLine). The limit says how much memory may a worker use at most for storing the objects it owns. The *rest* of memory below the limit will be used for the object cache. The memory limit is relative to the size of the distributed data (e.g., "100% memory" means that each worker is allowed to store the whole scene in its memory). The speedups in Figure 4 are relative to the computational time of a single worker with a warm start (no antialiasing). The hit cache ratio was above 99% and the total number of object requests about $3,000$ in the 20% memory case (only about 20% of all objects are relevant for rendering the image). The hit ratio dropped only by a promile in the 10% memory case—however, the total number of object requests grew to about $500,000$. In the 5% case, the hit ratio was about 85% and the total number of object requests about $7,000,000$.

Some data are missing in the graphs in Figure 4. In these cases the scene could not be stored in the distributed memory of worker processes due to the (simulated) memory limits.

5 Conclusions

A simple and robust parallelization of the ray tracing algorithm was presented which allows rendering of large scenes that do not fit into memory of a single processor.

An analysis of a load balancing strategy for image space partitioning was given. A parallel ray tracer implementing the described ideas was integrated as a component of a remote rendering system during the project HiQoS [16]. The proposed design of the distributed object database may serve as a basis for developing other parallel global illumination algorithms.

References

1. D. Badouel, K. Bouatouch, and T. Priol. Distributing data and control for ray tracing in parallel. *Computer Graphics and Applications*, 14(4):69–77, 1994.
2. R. Caubet, Y. Duthen, and V. Gaidrat. A tridimensional architecture for fast realistic image synthesis. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International 1988)*, pages 135–149. Springer-Verlag, 1988.
3. J. G. Cleary, B. Wyvill, G. M. Birtwistle, and R. Vatti. Multi-processor ray tracing. *Computer Graphics Forum*, 5(1):3–12, 1986.
4. M. Dippé and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *Computer Graphics (Proceedings of SIGGRAPH 1984)*, 18(3):149–158, 1984.
5. A. S. Glassner, editor. *Introduction to ray tracing*. Academic Press, Inc., 1989.
6. S. Green. *Parallel Processing for Computer Graphics*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1991.
7. S. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *Computer Graphics and Applications*, 9(6):12–26, 1989.
8. D. A. J. Jevans. Optimistic multi-processor ray tracing. In R. A. Earnshaw and B. Wyvill, editors, *New Trends in Computer Graphics (Proceedings of CG International 1988)*, pages 507–522, 1989.
9. M. J. Keates and R. J. Hubbold. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum*, 14(4):189–202, 1995.
10. H. Kobayashi, T. Nakamura, and Y. Shigei. A strategy for mapping parallel ray tracing into a hypercube multiprocessor system. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International 1988)*, pages 160–169. Springer-Verlag, 1988.
11. W. Lefer. An efficient parallel ray tracing scheme for distributed memory parallel computers. In *Proceedings of the Parallel Rendering Symposium*, pages 77–80, 1988.
12. M. L. Netto and B. Lange. Exploiting multiple partitioning strategies for an evolutionary ray tracer supported by DOMAIN. In *First Eurographics Workshop on Parallel Graphics and Visualisation*, 1996.
13. I. S. Pandzic, N. Magnenat-Thalmann, and M. Roethlisberger. Parallel raytracing on the IBM SP2 and T3D. *EPFL Supercomputing Review (Proceedings of First European T3D Workshop in Lausanne)*, (7), 1995.
14. P. Pitot. The Voxar project. *Computer Graphics and Applications*, pages 27–33, 1993.
15. T. Plachetka. POV||Ray: Persistence of vision parallel ray tracer. In L. Szirmay-Kalos, editor, *Proceedings of Spring Conference on Computer Graphics (SCCG 1998)*, pages 123–129. Comenius University, Bratislava, 1998.
16. T. Plachetka, O. Schmidt, and F. Albracht. The HiQoS rendering system. In L. Pacholski and P. Ruzicka, editors, *Proceedings of the 28th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2001)*, Lecture Notes in Computer Science, pages 304–315. Springer-Verlag, 2001.
17. T. Priol and K. Bouatouch. Experimenting with a parallel ray tracing algorithm on a hypercube machine. In *Proceedings of Eurographics 1988*, pages 243–259. Elsevier Science Publishers, 1988.
18. E. Reinhard and A. Chalmers. Message handling in parallel radiance. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 486–493. Springer-Verlag, 1997.
19. E. Reinhard, A. Chalmers, and F. W. Jansen. Hybrid scheduling for parallel rendering using coherent ray tasks. In *1999 IEEE Parallel Visualization and Graphics Symposium*, pages 21–28. ACM SIGGRAPH, 1999.

20. E. Reinhard, A. J. F. Kok, and A. Chalmers. Cost distribution prediction for parallel ray tracing. In *Second Eurographics Workshop on Parallel Graphics and Visualisation*, pages 77–90, 1998.
21. I. D. Scherson and E. Caspary. Multiprocessing for ray tracing: a hierarchical self-balancing approach. *The Visual Computer*, 4(4):188–196, 1988.
22. T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.