POV ‖ Ray

PERSISTENCE OF VISION PARALLEL RAYTRACER

Tomáš Plachetka
Department of Computer Graphics
Faculty of Mathematics and Physics
Comenius University
84215 Bratislava, Slovakia
e-mail: plachetka@fmph.uniba.sk

ABSTRACT

This paper is about design and implementation of POV ‖ Ray—an efficient parallel 3D rendering engine based on the popular (sequential) POV-Ray raytracer. Although distributed freely, thanks to its powerful scene description language, special effects capabilities, portability and output quality, POV-Ray can compete with commercial raytracers. POV ‖ Ray provides the same functionality as its sequential sibling, but runs approximately N times faster on a parallel machine with N processors (our experiments with up to 40 processors show almost a linear speedup). We used a loadbalanced screen subdivision technique to increase performance. Furthermore, we added two-pass solution and animation support to POV ‖ Ray. POV ‖ Ray uses GOLEM communication library based on PVM. GOLEM, as well as POV ‖ Ray and a parallel online radiosity engine with two-pass support, were developed at University of Paderborn (Germany) within project PARAGRAPH.

Keywords: raytracing, parallel, loadbalancing

1 INTRODUCTION

Although many speedup techniques are known today, raytracing still is a time consuming process. Parallelization is an attempt to make it faster. Former approaches to raytracing parallelization can be roughly divided into two groups [GREEN89]: techniques using screen space subdivision (each processor holds entire scene in its memory and computes a subset of screen pixels in parallel with other processors); and techniques using 3D object space subdivision (each processor holds only a part of the scene in its local memory, exchanging either rays or objects with its neighbours when needed).

The 3D space subdivision techniques ([DIPPE84], [SCHER88], [GREEN89], [PITOT93]) have been necessity in former times. The main reason was that the entire scene did not fit into memory of each processor in a distributed memory machine. Another reasons were ray coherence exploitation and a good loadbalancing behaviour reported by using these techniques.

Screen space subdivision techniques ([PRIOL89]. [WOODWARK84]) seem to be more natural (computations on screen pixels are independent, thus no communication between processors is needed), but have two drawbacks. Firstly, memory is not efficiently used (the entire scene is duplicated in processor memories if using a distributed memory machine), thus an unnecessary limitation is imposed to the maximum scene size. Secondly, it is not so obvious how to achieve a reasonable loadbalancing, because computational time on different pixels is unpredictable. Despite of these arguments, there is a very good software engineering reason why to use screen space subdivision—it is much easier to parallelize in this way an already existing raytracing program, reusing known sequential raytracing speedup techniques, rather than to develop and tune a more complicated one. Next section and our efficiency tests show that the loadbalancing problem is not as critical as it seems to be.

In case of distributed memory architectures, the need for storing the entire scene in memory of each processor leads to an unpleasant limitation to the maximum scene size. However, this problem can be solved by maintaining a distributed object database. One processor (or several processors) with large memory stores all objects in the scene and acts as a database server. The other processors, doing the raytracing calculations have a limited cache memory. If a processor needs an object which is not currently in its cache, it asks a database server to send it. The processor requesting an object must make sure that there is enough memory for storing the object—if the cache is already full, a victim (or victims) must be found among the objects in the cache to make enough space for the requested object. The caching strategy tries to predict which objects will be needed in a near future, in order to minimize the number of requests.

2 PARALLELIZATION

One of the aims of project PARAGRAPH was implementation of a parallel rendering engine combining raytracing and radiosity. For the reasons described above we decided to parallelize an existing sequential state-of-the-art raytracer, using screen subdivision technique. POV-Ray proved to be a good choice.

The main POV‖Ray design goals were:

- Scalability and efficiency (measured by speedup).
- Portability.
- Preserving POV-Ray features.
- Two-pass solution support.

A process farm is suitable for screen subdivision (see Fig.1). There is one master process, responsible for subdividing the 2D screen into nonoverlapping parts, assigning these parts to worker processes and for collecting results (pixel colours) from them. The worker processes are initially waiting until they receive a job from master (a subset of pixels to be computed). After receiving a job, a worker performs raytracing calculations on the pixels, sends the result back to the master process and waits for another job.

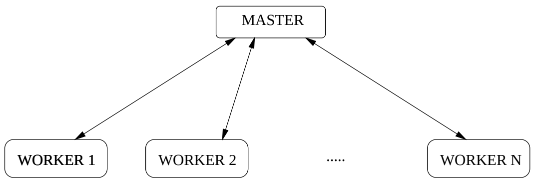

Figure 1: A process farm

This algorithm is quite straightforward. The crucial question is how to distribute the screen among workers.

2.1 Loadbalancing optimization

The simplest way how the master process can divide the screen among N workers is to cut it into N slices of equal sizes (Fig.2).

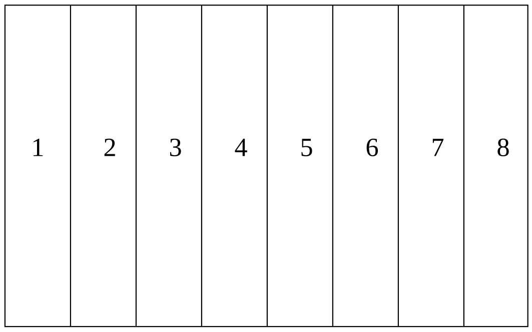

Figure 2: Simple work distribution for 8 workers

The disadvantage of this approach is possibly unequal loadbalancing. Different slices can take different time to compute. Here is an example of a bad case: Let slices 1–7 take 5 minutes to compute, whereas slice 8 takes 13 minutes. After 5 minutes of computation, workers 1–7 would be idle, while worker 8 would remain running for the next 8 minutes. During these 8 minutes only $\frac{1}{8}$ of the computational power would be used, and the total computation time would be 13 minutes. But the total time can be reduced to 6 minutes—by keeping all workers busy until the end of computation.

Generally we cannot predict in advance which parts of the screen are going to be computationally expensive. This means that there is no optimal static loadbalancing strategy ("static" means that all pixels are distributed among workers before the computation and this distribution is never changed).

Remark. In the following text we consider distributed memory and non-work-stealing model. (Once a worker is assigned a job, it cannot be interrupted until it finishes the computation of that job.) For interprocess communication we assume an asynchronous message passing model. •

Getting a perfect loadbalancing is very easy. The screen is subdivided into atomic parts (e.g. screen pixels, rows, columns) as shows Fig.3.

Indeed, the imbalance at the end of computation never exceeds the computational time of an atomic job. However, such a solution is inacceptable because of a high communication cost. This strategy creates too many small jobs, thus too many messages are exchanged between master and workers. The communication overhead can become even greater than the raytracing time.

A good loadbalancing algorithm should therefore fulfill two contradictory requirements: 1.minimum communication overhead; 2.minimum imbalance at the end of computation. A solution would be to assign large jobs to workers first, then smaller, yet smaller, . . . , finishing with atomic jobs. The key is to specify what "large" and "smaller" mean.

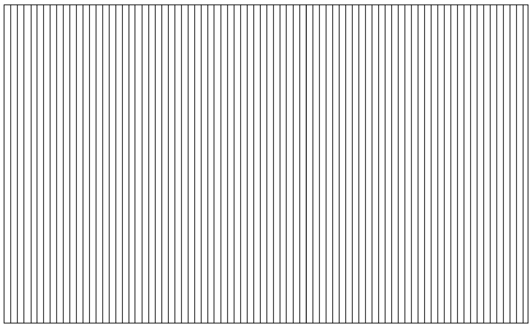

Figure 3: Another simple work distribution (atomic parts are screen columns in this case)

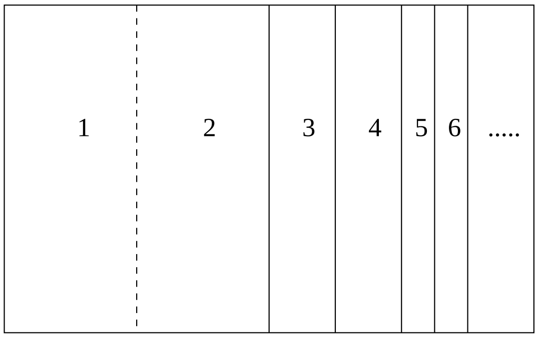

Figure 4: Illustration of assigning jobs in the perfect loadbalancing algorithm for $T = 3.0$, $N = 2$. Numbered parts are the jobs assigned to workers, in that order. (Jobs 1 and 2 are assigned before the while loop.)

Let us assume that for any two jobs of equal sizes

$$\frac{job_1 \ computational \ time}{job_2 \ computational \ time} \leq T,$$

where $T \geq 1.0$ is a constant known in advance. Given the constant T, the following algorithm ensures optimal loadbalancing, significantly reducing communication overhead (N denotes number of workers):

$WorkLoad$=$number\ of\ atomic\ parts$;
$Ratio = 1/(1 + T(N - 1))$;
$JobSize = (WorkLoad \cdot Ratio)$;
. . . assign a job of size $JobSize$ to each worker;
$WorkLoad = WorkLoad - N \cdot JobSize$;
while ($WorkLoad > 0$)
{
    . . . wait until an idle worker asks for a job;
    if ($JobSize > atomic\ job\ size$)
    {
        $JobSize = WorkLoad \cdot Ratio$;
    }
    . . . assign the worker a job of size $JobSize$;
    $WorkLoad = WorkLoad - JobSize$;
}

The algorithm is pessimistic. It assigns jobs just as large as possible, still ensuring no imbalance at every moment. Fig.4 illustrates how it works.

Remark. Note that the two simple job distribution strategies mentioned before are special cases of this algorithm. For $T = 1.0$ the screen is subdivided into N equal parts. For $T \longrightarrow \infty$ we get the second extreme, when the screen is subdivided into "infinitely" many "infinitely" small jobs (atomic jobs). •

Remark. As we found later, the same idea has been independently used in [PANDZ95] for shared memory model. •

Remark. Unfortunately, we usually do not know the constant T beforehand. T must be set empirically and for a specially constructed scene the loadbalancing might not be equal as promised. A value of $T = 2.5$ is suggested in [PANDZ95]. Our experiments (see section 6) justify this setting.

Moreover, the size of an atomic job must be chosen so that the time needed to transfer a message from worker to master plus the time needed for master to process the result does not exceed the computation time of an atomic job. The proper atomic job size may be dynamically computed by measuring the communication time between master and workers. However, in our experiments, an atomic job was always set to a single screen column—too large for ROSENTHALERHOF scene, and too small for SKYVASE scene (see section 6). This setting negatively influenced our speedup measurements for the mentioned scenes. •

2.2 Minimizing idle time in worker processes

If the number of workers is large and computation of one job does not take much time, then the master process becomes a bottleneck of the application. There can be several idle workers waiting for a job while master is still processing output of another worker.

Another potential source of similar problems may be slow communication between master and workers. Master and workers need not necessarily run on the same machine. Typically, the master process runs on a workstation with graphical capabalities, whereas workers run on a parallel computer. The communication within the parallel machine can be very fast, but the link between the workstation and the parallel machine can be slow. When a worker finishes computation of a job, it sends result to master—and remains idle while the result goes through the slow link to master, while master processes the message, and while another job specification is sent back through the slow link.

To minimize the idle time in workers, an additional process must have been inserted between master and workers. This process, called *loadbalancer*, takes over the responsibility of distributing jobs to workers. Loadbalancer is almost completely independent on master—it only needs to know the image size in pixels—so it can be placed "close to workers", on one of the processors in the parallel machine. (Of course, loadbalancer runs the loadbalancing algorithm described above.) When a worker finishes its computation of a job, it sends a short notification message to loadbalancer, then sends the result to master. Meanwhile loadbalancer receives the notification message and sends the worker a new job specification via the fast link.

2.3 Speeding up scene parsing

When working with large scenes, the preprocessing time needed to load the scene from disk to processors' memories must also be considered. This problem becomes critical when the worker processes run on a parallel disk-less machine connected to a disk server with a single communication link. If there are, say, 40 workers, each one reading 10MB scene from the disk, then 40x10MB=400MB must be sent over the link.

The solution is to read the scene from the disk only once, then broadcast it among workers. Broadcasting even a large message is usually much faster than disk operations involving the same volume of data.

Of course, reading the entire file by one process and subsequent broadcasting the huge message would not bring any improvement. A kind of buffering must be used. The input scene file is read by one process in small chunks. After reading a chunk, the process broadcasts the chunk to all workers, so that they can parse the chunk while another chunk is being read from the disk. Optimal chunk size is strongly system dependent and must be found empirically. (A chunk size of 64kB was a good choice for the environment we used for our experiments, see section 6.)

3 TWO-PASS SOLUTION

Original POV-Ray (since version 3.0) offers so called "radiosity" mode to approximate the ambient light by distributed raytracing. We shall describe a simple two-pass mechanism which POV‖Ray uses instead of distributed raytracing, allowing to combine raytracing with an already precomputed radiosity solution. (A similar method is suggested in [SHIRL90].)

For two-pass solution purposes we assume that the entire scene is modelled by triangles only. In the first pass the radiosity solution is computed. The output from the first pass are colour intensities stored in the vertices of all triangles. However, the radiosity algorithm is slightly modified—it subtracts the energy transfered to a triangle when shooting light source patches. Without this direct lighting, the radiosity output contains the indirect (ambient) lighting of all objects in the scene.

After the first pass, the scene is exported in POV‖Ray scene description language (POV‖Ray does not implement the radiosity pass). We extended the POV-Ray language with two new object types: *radiosity_triangle* and *radiosity_smooth_triangle*. There is only a slight difference between these types and already existing *triangle*s and *smooth_triangle*s. The radiosity triangles have the indirect lighting from the radiosity pass stored in their vertices.

The second pass is then running POV‖Ray with a special flag saying that it should handle the radiosity triangles a special way. When applying the lighting model to a point on a radiosity triangle, the constant ambient term is replaced by an interpolation of the triangle's vertex colours gathered from the radiosity pass.

4 IMPLEMENTATION

POV‖Ray uses GOLEM communication library. GOLEM is based on PVM, but provides a higher programming interface than PVM. A GOLEM application consists of two parts, called frontend and backend. Frontend typically runs on a graphics workstation, providing user interface. Backend typically runs on a parallel machine (this means any machine running PVM; frontend and backend may also run on a single workstation) and performs the computationally expensive calculations. All processes can communicate with each other. Two special GOLEM processes connect frontend and backend: system interface process (running on frontend) and controller (running on backend). When, say, a frontend process sends a message to a backend process, the message is sent over a socket to controller which translates the message into a PVM message and sends it to recipient. This layer is completely transparent to an application programmer. The application programmer only uses high-level GOLEM communication functions and does not see either PVM or the socket communication used between frontend and backend. (Andreas Dilger [DILGER] did a similar implementation, based on the pure PVM library.)

POV‖Ray starts N instances of the worker process plus the loadbalancer process on backend. All backend processes keep waiting, waiting for orders from master. Master is started on frontend as the last process, using the same commandline syntax and initialization files as the sequential POV-Ray program. After parsing its commandline and initialization files master controls the computation. The computation consists of the following stages:

- Initialization.
- Parsing the scene file.
- Raytracing.
- Termination.

Remark. Not all of the phases are fail-safe, e.g. parsing an incorrect scene file leads to abnormal termination. In such a case POV $\|$ Ray ensures a correct termination of the whole application (terminating all application processes). •

Both frontend and backend, as well as GOLEM library, are well portable to Unix-like systems running PVM. Backend was successfully tested on SUN workstation cluster (Solaris 2.5), Parsytec GCPP (parallel computer based on T805 transputers) and Parsytec CC-48nK (parallel computer based on Motorola Power PC processors). Frontend was successfully tested on SUN (Solaris 2.5), SGI (IRIX) and Parsytec CC-48 node (Motorola Power PC running AIX).

5 ANIMATION SUPPORT

Although time needed for parsing the scene file can be reduced as described in the section 2, this stage can still take lot of time when working with large scenes (especially the scenes preprocessed by radiosity tend to be large). When creating a two-pass animation in which only the camera properties change, it is highly desirable to render many frames without reparsing the scene after each frame (the way how the sequential POV-Ray does it). We introduced a simple and elegant solution to this problem. A special parameter is added to the master's commandline, telling the master to keep all backend processes running after rendering one frame (workers keep the scene in their memories). The master process terminates, but can reconnect to the application later. After the master reconnects, the workers parse only a very short scene file containing the new camera description and a new frame is rendered.

A typical usage of this feature is an interactive camera animation. A user "walks" through a scene precomputed by radiosity. The path is remembered and a sequence of camera positions interpolating the path is generated. This sequence can be used later for raytracing the same scene from the different viewpoints, producing a high quality two-pass animation.

6 EFFICIENCY TESTS

The following experiments illustrate the efficiency of POV $\|$ Ray implementation. We measured POV $\|$ Ray speedup on three scenes (BATH, ROSENTHALERHOF, SKYVASE). The size of all rendered frames was 640x480 pixels. Online graphical display was switched off. This was the backend hardware used in our experiments:

System: CC48 (Cognitive Computing)
Manufacturer: Parsytec Ltd, Germany
Nodes: 48

Node description:
Processor: Motorola Power PC 604
Speed: 133MHz
Memory: 64MByte
MIPS: 400
MFLOPS peak: 266
Links: 1
Link speed (peak): 1 Gbit/s

Out of the 48 nodes of CC48 only 44 can be used by an application's backend. We used up to 40 nodes in our experiments. Furthermore, when using more than 4 processors, two backend processes (GOLEM controller and the loadbalancing process) were mapped onto separate processors (e.g. there were 38 POV $\|$ Ray worker processes running when using 40 processors).

As frontend we used SUN Sparc 10 running Solaris 2.5, 80MB of memory. During the experiments almost nothing else has been running on the machine. We could not measure the network load between frontend and backend, but suppose that its influence was neglectable.

The tables below show POV $\|$ Ray performance on three test scenes. N denotes number of worker processes, T denotes the parameter of the loadbalancing algorithm from section 2 (T=1.0 means subdividing the screen into N parts of approximately equal sizes). The absolute time values are given is seconds.

Scene BATH (Fig.5) contains about 50000 (non-meshed) radiosity triangles and 16 light sources. File size: about 10MB. Average parsing time: about 153 seconds.

N	Raytracing time (Speedup)					
	T=1.0		**T=2.0**		**T=3.0**	
1	454	(1.0)	454	(1.0)	454	(1.0)
2	298	(1.5)	228	(2.0)	231	(2.0)
6	109	(4.2)	79	(5.7)	79	(5.7)
14	48	(9.5)	35	(13.0)	36	(12.6)
30	24	(19.0)	18	(25.2)	18	(25.2)
38	19	(23.9)	16	(28.4)	15	(30.3)

Scene ROSENTHALERHOF (Fig.6) contains about 86000 triangles (about 2200 meshes) and 12 light sources. File size: about 10MB. Average parsing time: about 182 seconds.

N	Raytracing time (Speedup)					
	T=1.0		T=2.0		T=3.0	
1	1768	(1.0)	1768	(1.0)	1768	(1.0)
2	934	(1.9)	886	(2.0)	886	(2.0)
6	364	(4.9)	298	(6.0)	303	(5.8)
14	174	(10.1)	129	(13.7)	129	(13.7)
30	92	(19.2)	62	(28.5)	62	(28.5)
38	73	(24.2)	49	(36.0)	50	(35.4)

Scene SKYVASE (Fig.7) is a very simple scene from POV-Ray package. It contains just a few curved objects. The file size is only 2kB and the file was always parsed within two seconds. We used this scene to see how POV$\|$Ray behaves on small scenes like this one.

N	Raytracing time (Speedup)					
	T=1.0		T=2.0		T=3.0	
1	251	(1.0)	251	(1.0)	251	(1.0)
2	134	(1.9)	127	(2.0)	127	(2.0)
6	51	(4.9)	43	(5.9)	42	(6.0)
14	30	(8.4)	19	(13.2)	19	(13.2)
30	24	(10.5)	14	(18.0)	13	(19.3)
38	17	(14.8)	20	(12.5)	14	(18.0)

7 CONCLUSIONS

In the paper we presented POV$\|$Ray—an efficient parallel raytracing engine. The parallelization problems and techniques summarized in section 2 apply generally.

POV$\|$Ray efficiency tests show almost a linear speedup for up to 40 processors. For the ROSENTHALERHOF scene, the atomic job size setting of a single screen column was too large and subsequent uneven loadbalancing degraded the performance. The reason for the abnormal speedup behaviour in case of SKYVASE scene was caused again by an inappropriate atomic job size setting—for such a simple scene the time needed for processing incoming results in the master process approached the computation time of raytracing performed by workers (in this case the atomic job size setting of a single screen column was too small and the master process became a bottleneck of the application). Both artefacts can be eliminated by a dynamical setting of the atomic job size in our loadbalancing algorithm (see the last remark in section 2.1). According to our experiments, the setting of constant $T = 2.5$ in the loadbalancing algorithm suggested by [PANDZ95] seems to be a reasonable value.

Future work can include looking for a better loadbalancing strategy. The problem arising when an entire scene does not fit into a processor's memory can be addressed by either looking for a better scene representation suitable for both radiosity and raytracing, or by investigation of efficient caching strategies for a distributed object database combined with screen subdivision parallel ratracing techniques.

8 ACKNOWLEDGEMENTS

We would like to express thanks to the people working on the project PARAGRAPH: Olaf Schmidt, Ludger Reeker, Jan Rathert, Joerg Hundertmark; and also to other people at University of Paderborn, whom it has been a pleasure to work with.

9 FIGURES

Figure 5: Scene BATH

Figure 6: Scene ROSENTHALERHOF

Figure 7: Scene SKYVASE

References

[DILGER] Andreas Dilger: *PVM Patch for POV-Ray*, http://www-mddsp.enel.ucalgary.ca /People/adilger/povray/pvmpov.html

[DIPPE84] M.Dippé, J.Swenssen: *And adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis*, Computer Graphics, Vol.18, Nr.3, 1984.

[GREEN89] S.A.Green, D.J.Paddon: *Exploiting Coherence for Multiprocessor Ray Tracing*, IEEE Computer Graphics and Applications, 1989.

[PRIOL89] T.Priol, K.Bouatouch: *Static Load Balancing for Parallel Ray Tracing*, The Visual Computer, March 1989.

[PANDZ95] I.S.Pandzic, N.Magnenat-Thalmann: *Parallel raytracing on the IBM SP2 and T3D*, Miralab, University of Geneva, http://sawww.epfl.ch/ SIC/SA/publications/SCR95/ 7-95-54a.html, 1995.

[PITOT93] P.Pitot: *The Voxar Project*, IEEE Computer Graphics and Applications, 1993.

[SCHER88] I.D.Scherson, E.Caspary: *Multiprocessing for Ray Tracing: a Hierarchical Self-Balancing Approach*, Visual Computer 4, Springer-Verlag, 1988.

[SHIRL90] P.Shirley: *A ray traced method for illumination calculation in diffuse-specular scene*, Proceedings of Graphics Interface, 1990.

[WOODWARK84] J.R.Woodwark: *A Multiprocessor Architecture for Viewing Solid Models*, Display Technology and Applications, Vol.5, No.2, 1984.