A PVM-based parallel implementation of the REYES image rendering architecture

Oscar Lazzarino Andrea Sanna Claudio Zunino Fabrizio Lamberti

Dipartimento di Automatica e Informatica
Politecnico di Torino Corso Duca delgi Abruzzi, 24
I-10129 Torino, Italy
oscar.lazzarino@tiscali.it, {sanna,c.zunino,lamberti}@polito.it

Abstract. In this paper a PVM-based distributed platform of the well known REYES rendering architecture developed by Pixar is presented. This work effectively tackles issues related to load balancing and memory allocation by a master-slave paradigm. In particular, the rendering is distributed performing both an image and an object space subdivision; in this way, low memory resources are necessary to the slave side. Examples show the effectiveness of the proposed work.

1 Introduction

Several algorithms are able to produce very realistic synthetic images, but their main drawbacks are the high computational times and the memory requirements. Although CPUs have enormously increased their computational capacity and the cost of memory chips has decreased, single processor solutions are often not able to provide the rendering of a scene within acceptable times. On the other hand, distributed low-cost architectures based on PC clusters have been proved to be effective and performing solutions.

This paper proposes a distributed implementation based on PVM [1] of the well-known REYES architecture [2] developed by Pixar. Although REYES is not able to directly take into account global illumination effects, reflection and transparency can be simulated in order to obtain a high level of realism.

Two main issues have to be addressed: load balancing and memory occupation. The first issue can be addressed performing a dynamic load balancing, that is, the image is split into a set of regions dynamically assigned to computation nodes (image space subdivision). An effective memory usage can be obtained dynamically providing every node only of primitives strictly necessary to render the region of image under analysis (object space subdivision).

The proposed solution performs a hybrid solution that merges image and object space subdivision, providing both an effective dynamic load balancing by means of a master-slave paradigm and an efficient memory management that avoids the duplication of the entire database on every slave node.

The paper is organized as follows: Section 2 reviews the main approaches involved in distributed and parallel rendering, while Section 3 briefly presents the

sequential version of the algorithm considered in this work. Section 4 describes the details of the parallel implementation and Section 5 shows results obtained testing the proposed architecture over a Gigabit-Ethernet PC cluster.

2 Background

High quality rendering algorithms always need high computational capacity and parallel approaches have been often used in order to reduce rendering times.

Independently of the rendering algorithm, two main strategies can be identified: image space subdivision-based and object space subdivision-based.

Image space subdivision-based algorithms split the image plane into several regions, and each processor has to compute one or more of them. With this approach the load distribution can be either static or dynamic; in the first case, each processor has a priori knowledge of pixels to be rendered, while, in the second case, the load is assigned at run time. Image space subdivision-based algorithms may suffer from two main drawbacks: the implementation may be critical on shared-memory machines, since a large number of accesses to memory may become a bottleneck (in massively parallel computation), and the whole scene database has to be duplicated at each processor when distributed-memory machines are employed.

The object space subdivision-based methods were designed to address the problem of the implementation of parallel algorithms on distributed memory machines. In such computers each processor has a small amount of local memory, so that each processor has to load into its memory a small subset of the entire database. Each processor checks the objects stored in its local memory and results are sent to the other processors by messages. The strategy of the database subdivision is very important because it affects the workload distribution among processors; the object distribution can be either static or dynamic. In the former case, each processor has a priori knowledge of the elements to be loaded into its memory, while in the latter case the objects are assigned at run time.

Indeed, a lot of work is known in the literature concerning the parallelization of ray-tracing and radiosity algorithms.

Ray-tracing is able to provide photo-realistic rendering by considering global illumination effects such as transparency and reflection. Moreover, the computation of a pixel is independent of the others, therefore, a straightforward parallelization can be obtained [3–12]. Of particular interest are the distributed rendering algorithms tailored for PC/workstation clusters. For instance, [13] and [14] proposed two distributed implementations of the well-known POV-Ray ray tracer [15] based on PVM.

Radiosity methods produce highly realistic illumination of closed spaces by computing the transfer of light energy among all of the surfaces in the environment. Radiosity is an illumination technique, rather than a complete rendering method. However, radiosity methods are among the most computationally intensive procedures in computer graphics, making them an obvious candidate for

parallel processing. Many of the parallel radiosity methods described in the literature attempt to speed up the progressive refinement process by computing energy transfers from several shooting patches in parallel (i.e., several iterations are performed simultaneously) [16–18].

At our knowledge, a new proprietary parallel version of the REYES architecture will be included in Pixar's RenderMan® R11 which will be released in Q4, 2002.

3 The REYES architecture

The REYES architecture can be viewed as a pipeline through which the data flows independently and therefore it can be classified as a local illumination system. This fact allows the rendering of scenes of arbitrary complexity; global effects as shadows, reflections, and refractions cannot directly be computed by the algorithm, but various methods exist to overcome this restriction.

The design principles behind REYES are discussed in depth in [2]; in this section, a brief outline of the algorithm will be only described. The algorithm is summarized in Fig. 1. The algorithm uses a z-buffer to remove hidden surfaces. The primitives in the input file are read sequentially and for each one a bounding box is built. This bounding box is used to determine whether the primitive is visible or not. If the bounds are completely outside the viewing frustum, the primitive is culled.

After these preliminary checks, the primitive is repeatedly split until it can be safely *diced* into a grid of *micro-polygons*. A micro-polygon is a little (not bigger than a pixel) quadrilateral polygon. The dicing operation is performed when the evaluation of the resulting grid shows it will not contain too many micro-polygons and their size will be not too different.

The grid is then shaded and broken into independent micro-polygons. Every single micro-polygon is then bounded, and if the bounding box happens to be entirely out of the the viewing frustum, the micro-polygon is discarded. Visible micro-polygons are then sampled one or more times per pixel, and the samples are checked against the z-buffer.

When there is no more data in the input file, the samples are then filtered to remove aliasing artifacts, dithered, and quantized and the final image is produced.

4 Parallel implementation

The proposed algorithm is based on the REYES architecture described in Section 3. To design a distributed rendering environment, the original architecture has been modified following a master-slave approach as shown in Fig. 2. The system consists of a master process and of several slave processes which actually implement the REYES algorithm on a subset of the input data. It is up to the master to split the original data set into smaller subsets and to dispatch

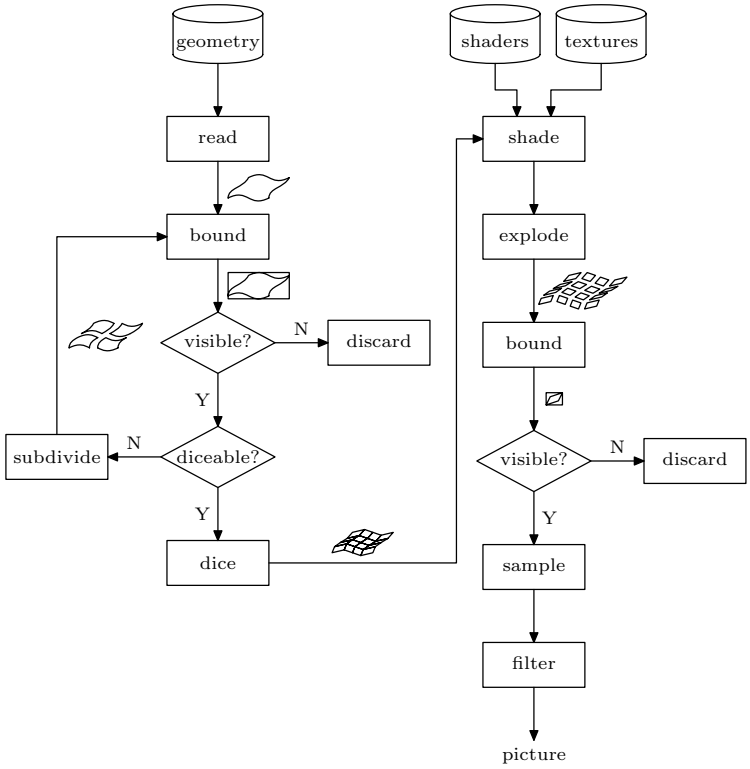

Fig. 1. Flow of data in the REYES pipeline

them to slave processes in order to efficiently distribute the computational load among available cluster nodes. This goal is achieved dividing the image area in rectangular blocks, named *buckets*.

The geometric primitives generating the scene are assigned to the bucket containing the top-left corner of the primitive bounding box, as shown in Fig. 3. Buckets are then submitted to slave processes where the rendering of the corresponding image block takes place. It has to be remarked that this bucket selection criteria assigns primitives covering adjacent blocks to a single bucket thus forcing the designated slave to process data which could be outside of its scope. It will be shown how the slave identifies those primitives falling out of the block and passes them back to the master process that will dispatch them to the proper slave.

4.1 Distributed algorithm description

In the following, the steps of the parallel rendering algorithm are presented and PVM master-slave message passing details are outlined:

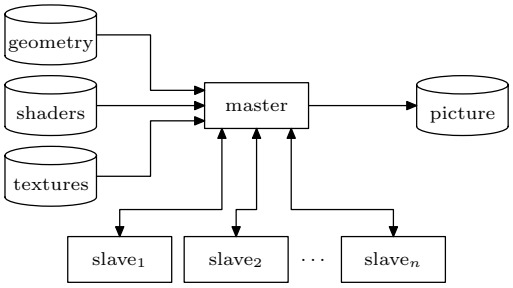

Fig. 2. Master-slave architecture

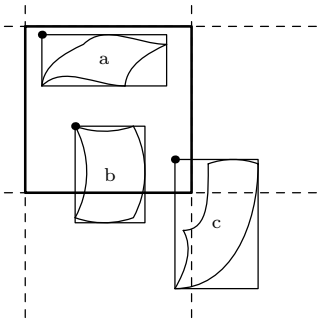

Fig. 3. Buckets subdivision

M→S the master parses the input file describing the scene and performs primitive-to-bucket assignments; then the master runs the slave processes and sends them an initialization message INIT containing camera, image and rendering algorithm related information including camera position and orientation, image resolution, buckets size, number of samples per pixel, micro-polygons grid size, shading rate, reconstruction filter, quantization, dithering and clipping planes parameters;

M←S when the slave is ready to process a bucket, it sends an ASK_BUCKET to the master;

M→S the master selects a bucket from the scheduling queue according to the priority parameter (see section 4.2) and sends to the slave a BUCKET message containing bucket coordinates;

M←S the slave performs several initialization operations (i.e. clears the z-buffer and the frame buffer) and sends the master an ASK_PRIMS message to request primitives;

M→S the master sends the slave a PRIMS message containing a set of primitives and micro-polygons for the current bucket in a serialized format (in this way, both an image and an object space subdivision is performed);

M←S the slave processes primitives and micro-polygons according to REYES algorithm. As previously mentioned, the REYES split operation can generate primitives falling outside the bucket bounds. These primitives are stored in a primitives temporary queue. Primitives that pass the diceable-test are diced into a micro-polygons grid and shaded. Micro-polygons corresponding to primitives partially outside bucket bounds are stored in a micro-polygons temporary queue. When the size of the primitives or micro-polygons queues reach a particular threshold value, a JUNK message containing all the data in the temporary queues is sent to the master. The master delivers received data to the appropriate buckets, according to the new bounding boxes;

M↔S when the primitives have been processed, the slave sends additional ASK_PRIMS messages to the master. The master replies with additional PRIMS messages and sends an EOB (End of Bucket) when there is no more data in the bucket;

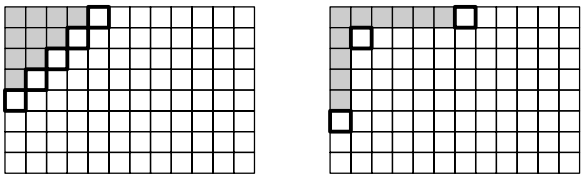

Fig. 4. Buckets to slave scheduling: the gray buckets are completed, while highlighted ones are ready to be scheduled

M←S the slave performs sample filtering operations, as well as gamma correction, dithering and quantization, and sends the master a RESULT message containing the processed image portion. Moreover, an ASK_BUCKET message notifies the master that the slave is ready to receive a new job;

M→S the master can reply by sending a new bucket or by repeating the EOB message indicating all buckets have been processed and the slave can stop its execution.

Slave processes are started and terminated by the master and only master-slave communications are allowed. It is worth mentioning that the master is the only process needing a direct access to input data (i.e. scene description, shaders and, textures), as shown in Fig. 2.

4.2 Load distribution

A key aspect of the proposed parallel rendering architecture is the job distribution strategy. The choice of the bucket-to-slave allocation algorithm deeply influences performances. This is mainly due to the fact that, while processing a bucket, the slave can encounter primitives which should be assigned to adjacent blocks. This flow of primitives between neighboring buckets always proceeds in the right-to-left and up-to-down directions, as clearly shown in Fig 3. Therefore, bucket dispatch to slave processes must follow well-defined scheduling rules. In particular, a bucket $B_{i,j}$ is ready to be scheduled only if both buckets $B_{i-1,j}$ and $B_{i,j-1}$ have been completely processed. When the master is started, only bucket $B_{0,0}$ is ready and it is assigned to a slave. As new buckets become ready, the master inserts them into a scheduling queue. The best performance can be obtained by trying to maximize the number of ready buckets.

It can be shown that this goal can be achieved by assigning to each bucket a queue extraction priority given by the sum of row and column indexes, thus forcing the scheduling algorithm to proceed in a diagonal direction. Fig. 4(a) shows that, by adopting the optimal scheduling scheme, when ten buckets have been processed, there are five buckets ready to be dispatched to slave processes. In Fig. 4(b) a sub-optimal scheduling scheme is presented: as in Fig. 4(a), ten buckets have already been processed but there are only three ready buckets.

Fig. 5. Three test scenes

The optimal case is not always possible, since the buckets can require different amount of time to be computed. Nevertheless, with the adopted strategy, the master manages to keep all the slaves busy for most of the time.

5 Examples and remarks

The system has been tested with three different input scenes, chosen for their characteristics. The "frog scene" contains many geometric primitives, but most of them are small in size. This implies that most primitives are entirely enclosed in a single bucket, and so the traffic of discarded primitives and micro-polygons is quite reduced. The "teapot scene" is constituted of only 32 geometric primitives, but most of them (especially the ones in the body of the teapot) are very large, and cover a great number of buckets. This causes the traffic between the master and the slaves to increase. In the third example the object teapot is replicated in order to fit all the space of the scene.

Graphs in Fig. 6 show the obtained speedup factors for the three scenes, rendered at a resolution of 3200×2400 pixels, with buckets of 256×256 pixels each. The tests were run on a cluster of seven PCs running Linux and PVM 3.4.4, each equipped with a 1400 MHz Athlon CPU, 512 MB of RAM, and a Gigabit Ethernet network adapter. A Gigabit Ethernet switch was used to connect the cluster nodes.

As expected, the frog test leads to a slightly better speedup than the teapot test. Moreover, many teapots provide better speed up factors than a single teapot as slaves are involved in larger computational loads. The overall performance of the system is satisfactory, with a mean speedup around 0.7 to 0.75. The image reconstruction and the bucket assignment parts are intrinsically sequential and cannot be parallelized and these tasks affect the overall performance.

Some extra efficiency can be gained allowing some slaves to work on "non-ready" buckets. With little changes, this would allow to pass data directly between slaves, relieving the master from some of its work. Another improvement for the algorithm could be the use of MPI architecture with the non-blocking receive calls.

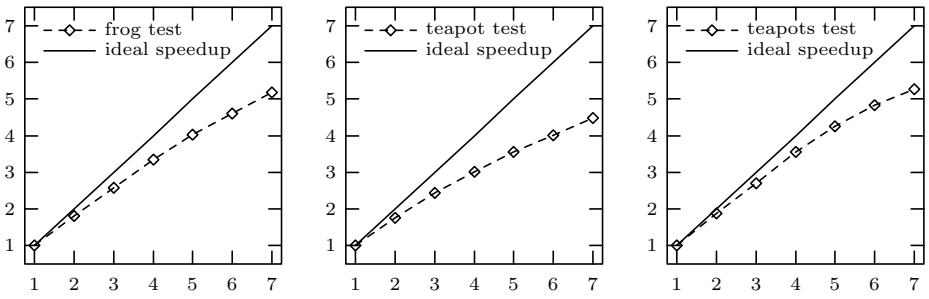

Fig. 6. Speedup factors with different number of slaves for the three test scenes shown in Fig. 5

6 Conclusion and acknowledgements

This paper proposes an effective distributed implementation of the REYES architecture providing an attractive alternative to other parallel rendering algorithms such as ray tracing and radiosity. The system implements a hybrid solution able both to dynamically distribute the computational load and to avoid unnecessary database duplications.

This project is supported by the Ministero dell'Istruzione dell'Università e della Ricerca in the frame of the Cofin 2001 project: elaborazione ad alte prestazioni per applicazioni con requisiti di elevata intensità computazionale e vincoli di tempo reale.

References

1. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V.: PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press (1994).
2. Cook, R.L., Carpenter, L., and Catmull, E.: The REYES image architecture. Proc. Siggraph, ACM Comput. Graphics Proceedings, **21** No. 4 (1987) 95–102.
3. Lin, T.T.Y., and Slater, M.: Stochastic Ray Tracing Using SIMD Processor Arrays. The Visual Computer, **7** No. 4 (1991) 187–199.
4. Plunkett, D.J., and Bailey, M.J.: The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed. *IEEE* CG&As, **5** No. 8 (1985) 52–60.
5. Dippé, M., and Swensen, J.: An Adaptive Algorithm and Parallel Architecture for Realistic Image Synthesis. Proc. Siggraph, ACM Comput. Graphics, **18** No. 3 (1984) 149–158.
6. Kobayashi, H., Nishimura, S., Kubota, H., Nakamura, T., and Shigei, Y.: Load Balancing Strategies for a Parallel Ray-Tracing System based on Constant Subdivision. The Visual Computer, **4** No. 4 (1988) 197–209.
7. Priol, T., and Bouatouch, K.: Static Load Balancing for Parallel Ray Tracing on a MIMD Hypercube. The Visual Computer, **5** No. 1-2 (1989) 109–119.
8. Green, S.A., and Paddon, D.J.: Exploiting Coherence for Multiprocessor Ray Tracing. *IEEE* CG&As, **9** No. 6 (1989) 12–26.

9. Reinhard, E., and Jansen, F.W.: Rendering Large Scenes Using Parallel Ray Tracing. Parallel Computing, **23** No. 7 (1997) 873–885.
10. Scherson, I.D. and Caspary, E.: Multiprocessing for Ray Tracing: a Hierarchical Self-balancing Approach. The Visual Computer, **4** No. 4 (1988) 188–196.
11. Yoon, H.J., Eun, S., and Cho, J.W.: Image parallel Ray Tracing Using Static Load Balancing and Data Prefetching. Parallel Computing, **23** No. 7 (1997) 861–872.
12. Sanna, A., Montuschi, P., and Rossi, M.: A Flexible Algorithm for Multiprocessor Ray Tracing. The Computer Journal, **41** No. 7 (1998) 503–516.
13. Dilger, A.: PVM Patch for POV-Ray. `http://www-mddsp.enel.ucalgary.ca/People/adilger/povray/pvmpov.html`
14. Plachetka, T.: POV‖Ray: Persistence of Vision Parallel Raytracer. Proc. of SCCG'98, (1998).
15. POV-Ray, `http://www.povray.org/`
16. Recker, R.J., George, D.W., and Greenberg, D.P.: Acceleration Techniques for Progressive Refinement Radiosity. Computer Graphics, **24**, No. 2 Proceedings of the 1990 Symposium on Interactive 3D Graphics, (1990) 59–66.
17. Feda, M., and Purgathofer, W.: Progressive Refinement Radiosity on a Transputer Network. Photorealistic Rendering in Computer Graphics: Proceedings of the 2nd Eurographics Workshop on Rendering, (1991), 139–148.
18. Chalmers, A.G., and Paddon, D.J.: Parallel Processing of Progressive Refinement Radiosity Methods. Photorealistic Rendering in Computer Graphics: Proceedings of the 2nd Eurographics Workshop on Rendering, (1991), 149–159.