Yugoslav Journal of Operations Research
16 (2006), Number 1, 125-135

AN IMPLEMENTATION OF RAY TRACING ALGORITHM FOR THE MULTIPROCESSOR MACHINES

Aleksandar B. SAMARDŽIĆ
Faculty of Mathematics,
University of Belgrade, Serbia and Montenegro
asamardzic@matf.bg.ac.yu

Dušan STARČEVIĆ
Faculty of Organizational Sciences,
University of Belgrade, Serbia and Montenegro
starcev@fon.bg.ac.yu

Milan TUBA
Faculty of Mathematics,
University of Belgrade, Serbia and Montenegro
auba@matf.bg.ac.yu

Received: November 2004 / Accepted: February 2005

Abstract: Ray Tracing is an algorithm for generating photo-realistic pictures of the 3D scenes, given scene description, lighting condition and viewing parameters as inputs. The algorithm is inherently convenient for parallelization and the simplest parallelization scheme is for the shared-memory parallel machines (multiprocessors). This paper presents two implementations of the algorithm developed by the authors for alike machines, one using the POSIX threads API and another one using the OpenMP API. The paper also presents results of rendering some test scenes using these implementations and discusses our parallel algorithm version efficiency.

Keywords: Computer graphics, Ray tracing, parallelization, multiprocessors.

1. INTRODUCTION

Ray Tracing is an advanced image generation algorithm ([18]). The algorithm consists of the two phases. First phase concerns the visible surface determination. During this phase, imaginary rays are traced from the viewpoint through the various points on the projection plane and intersected with all objects in scene. The closest intersection that is in front of the viewpoint determines the visible object along this ray. Second phase of the algorithm is conducted then, in order to calculate the illumination at given intersection point. In that order, rays are traced from the intersection point to each light source in scene. If alike ray is intersecting some object in scene before reaching the corresponding light source, then the intersection point is in shade regarding given light source. Thus, this light source is giving no contribution to overall illumination in given point. Otherwise, local illumination equation ([4]) is applied with regard to given light source and its contribution is added to illumination amount in the intersection point. Further, if material surface is reflective and/or transparent, reflection and refraction rays are traced in reflection and refraction directions in order to estimate global illumination influence in given point.

Because of applying the global illumination model, Ray Tracing algorithm is capable to generate much more realistic and attractive images than Z-Buffer algorithm and other algorithms presently used for the real-time 3D graphics. However, Ray Tracing is also, even with applied efficiency schemes, rather slow in comparison with alike algorithms and thus inappropriate for the real-time rendering. Because of this, since invention of Ray Tracing algorithm there was strong incentive to speed-up the calculation and after crucial algorithm improvement options exercised (through applying mentioned and other less used efficiency schemes), the parallelization remained as only viable solution.

Parallelization of the Ray Tracing algorithm for the multiprocessor machines is an area that was not much researched. On the other side, alike machines are since recently commonly available and that was our motivation to approach a parallel implementation of the algorithm for the multiprocessor machines. We expected to confirm that similar implementation is reachable and efficient. We also expected to collect some measuring and analyze them in order to be able to point to optimally structure parallel implementation. As an aside goal, we expected also to compare different known sequential Ray Tracing efficiency schemes regarding their behavior under multiprocessor parallelization.

The rest of this paper is organized as follows: section 2 presents sequential Ray Tracing algorithm; section 3 analyzes previous work regarding parallelization of Ray Tracing algorithm; section 4 outlines a parallel implementation of the algorithm for the shared memory parallel machines; section 5 presents the results obtained by the parallel version of the algorithm and makes comparison of sequential and parallel algorithm versions performance. Finally, Section 6 presents the conclusions.

2. SEQUENTIAL ALGORITHM

This section more closely examines sequential Ray Tracing algorithm, in order to be able to detect and develop parallelization approach. Rays traced from the viewpoint are usually denoted primary rays, while rays traced towards light sources and in reflection and refraction directions are collectively denoted secondary rays. Illumination coming from reflection and refraction rays is calculated recursively, on the same manner as for primary rays. This illumination is then multiplied with corresponding reflection and refraction factors and added to local illumination calculated for given intersection point, thus giving total amount of illumination in this point. This amount of illumination then determines final color of image pixel corresponding to the primary ray.

An example of the procedure is depicted in Figure 1.

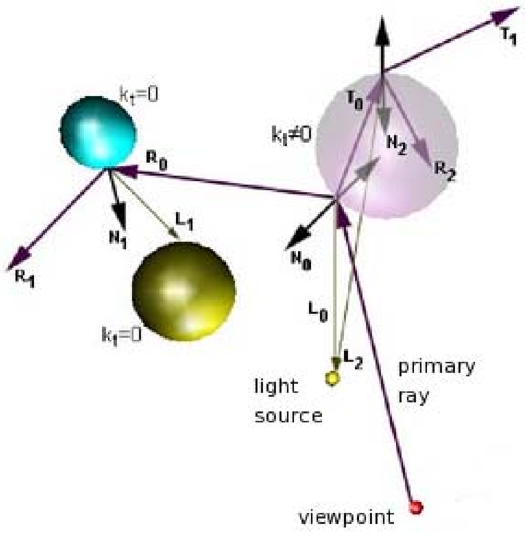

Figure 1: Recursive Ray Tracing procedure

The scene consists of three spheres, one transparent in the upper right part of the figure and two opaque in the left part of the figure. There exists one light source in the scene. Primary ray corresponding to some pixel is traced and found so that it intersects with transparent sphere. Now, the light ray L_0 is traced from the intersection point to the light source. The ray L_0 is not intersecting any object in the scene before reaching the light source, thus the intersection point is not in shadow with regard to the light source and local illumination model is applied giving local illumination in the intersection point coming from the light source. Since the sphere surface is both reflective and transparent, reflection ray R_0 and refraction ray T_0 are traced, too. Directions of reflection and refraction rays are determined by the well-known optical laws established by Fresnel and Snell respectively. The illumination coming along these rays is calculated recursively. For example, the ray R_0 is intersecting second sphere, sitting in top left corner of the figure. The light ray L_1 is traced from new intersection point towards the light source. However, this ray is intersecting third sphere before reaching the light source, thus there is no direct contribution from the light source to the illumination in this intersection point. But the intersected sphere is reflective, thus new ray R_1 is traced in the reflection direction in order to calculate the illumination coming from this direction. The illumination calculated in this manner is multiplied by corresponding reflection coefficients of second and first (transparent) sphere and added to the illumination of the intersection point of the primary ray and first sphere. The same procedure applies for the illumination calculated for the refraction ray T_0. Thus, the total amount of the illumination in the intersection point of the primary ray and first sphere is accumulated by this recursive procedure and finally this amount of color is assigned to the corresponding pixel.

The important issue in described procedure is of course the recursion termination criteria. One possibility is that recursion is limited to a fixed depth, usually 5 levels of recursion. Better solution is an adaptive recursion depth. Since reflection and refraction coefficients are values in [0,1] range, multiplying the illumination amounts with these coefficients while propagating the rays is usually causing fast decrease of the influence of the illuminations calculated in higher recursion depth. Therefore, further recursion could be avoided without noticeable impact on the final image when the coefficients product along some recursion path decreases below a predefined value.

Another issue in described procedure is the efficiency. When having intersecting rays with the scene object as the fundamental algorithm step, it is of utmost importance to have these intersecting implemented as fast as possible. Two different efficiency schemes are devised for improving the speed of this aspect of the algorithm:

1. Bounding volume hierarchies ([10]), where each object in the scene is bounded by the corresponding bounding volume and the hierarchy of these volumes is created. Boxes are usually used as the bounding volumes, because the cost of intersecting a ray with a bounding volume must be lower than intersecting a ray with any of objects in scene, and intersecting a ray with a box is very fast. The procedure is also devised for the automatic creation of best (with the lowest cost with regard to the intersection procedure) hierarchies ([7]). Each ray is then intersected with the hierarchy nodes. The objects are stored in the hierarchy leafs and a ray is intersected with them only if leafs reached. When a ray is not intersecting with a node upper in the hierarchy, that means that the ray is missing all nodes and leafs and the corresponding sub tree. Thus, the significant savings are achieved because the ray is not directly intersected with many objects in scene.
2. Voxel grids, where the scene bounding volume is divided into 3D cells (voxels) of same size and for each voxel all primitives containing at least part of this voxel are enumerated. A ray is then traced against the grid, using extended version of the 2D DDA algorithm ([1]). Each time when next voxel traversed by the ray is determined, the ray is intersected with all objects enumerated for that voxel (of course, if not already intersected with the given object during traversal of some earlier voxel). The idea is here to intersect a ray with more promising objects (objects that are close to the ray path and also close to the ray origin) earlier and thus again to avoid intersecting with many of objects in scene.

3. PREVIOUS PARALLELIZATION WORK

Lots of the early Ray Tracing algorithm parallelization work was directed to the parallelization trough implementing the algorithm on specific parallel machines ([3], [5], [9]). Some work is even invested in designing processors and architectures dedicated for Ray Tracing ([13]). However, because of the specific nature of this work, this kind of research was not broadly applicable and thus later efforts were more focused on the computational nature of the algorithm and more generic parallelization models.

Data-oriented parallelization was most often used parallelization model for Ray Tracing algorithm ([11]). With this model, the scene database is divided across processors. Each processor is performing all calculations related to assigned scene sub-domain. When a ray is leaving a sub-domain assigned to the given processor, appropriate control message is generated and passed to the processor owning entering sub-domain for further processing. Both bounding volume hierarchies and voxel grids as efficiency data structures are convenient for alike arrangement, but voxel grids have clear advantage that statistically each part of grid has the same probability of intersecting with a ray, while with bounding volume hierarchies there is much more probability of being intersected with rays for the nodes upper in hierarchy.

Alternative parallelization model for Ray Tracing algorithm is the control-oriented parallelization. With this model, the scene database is residing in the shared memory and is thus accessible to each processor. This parallelization model was not as thoroughly researched as data-oriented parallelization primarily because the shared-memory parallel machines were not commonly available as it was the case with the distributed memory parallel machines, which are targeted by former model. However, while new developments are still offering very interesting ideas for the data-oriented parallelization ([6]), wide availability of the shared-memory machines makes very appealing to examine Ray Tracing algorithm implementations crafted for this type of parallel machines and one alike implementation is discussed in this paper.

4. IMPLEMENTATION DETAILS

Since at the moment no commonly accepted public domains or any other type of supporting libraries for Ray Tracing algorithm exist, we had to develop our Ray Tracer from scratch. First step during our implementation of Ray Tracing algorithm for multiprocessor machine was implementation of sequential version of the algorithm. The decision was made for the NFF (Neutral File Format) input file format, generated by the SPD (Standard Procedural Database) software ([8]). The SPD software is capable to generate dozen of the procedural scenes that are used as standard scenes for benchmarking rendering algorithms. Further, the scenes are generated in mentioned NFF format that is very easy to parse, so that the render writer could concentrate on the render and not on the input format intricacies (that is not so often case with other 3D file formats).

The implementation of the render is conducted in C programming language, but still according to the principles of the object-oriented design. This was achieved by strictly following defined set of the object-oriented C programming practices ([16]) and basic principles of the object-oriented analysis and design ([2]). The sequential render implementation supports both mentioned efficiency schemes (the bounding volumes hierarchy and the voxel grid) and is comparable in the performance with the popular public domain Ray Tracing software (like POVRay) regarding rendering the SPD test scenes.

After having completed the implementation of the sequential render, we approached parallelization. Most often used methods of the parallelization for the shared memory parallel architecture are different threads mechanisms. Since threads API is standardized through the POSIX standard, POSIX threads are selected for implementation. However, another programming paradigm has recently emerged for the parallel programming on the shared memory parallel machines and this is the OpenMP mechanism, so we added an OpenMP based implementation too to our render. We named the render PARRT (**PAR**allel **R**ay **T**racer) and made its source code publicly available from `http://www.nongnu.org/parrt/`. The selection between the POSIX threads and the OpenMP parallel implementation is a compile time option.

The POSIX threads API ([14]) is defined as a set of C language programming types and procedure calls. All threads within a process share the same address space and this is how the shared memory paradigm is supported by this API. Very convenient feature of Ray Tracing algorithm is that there exists no inherent possibility for shared memory write conflicts during the rendering. Namely, most of the algorithm memory operations are read accesses and single write access is for storing calculated pixel color into the final image representation in memory. But even there, each pixel has its own memory location and if no two processors have assigned the same pixel for the calculation (and there exist no reasons for alike meaningless duplication of work), there is no possibility for conflicts even for this operation. In such a way, it is relatively simple to adapt the sequential implementation for the POSIX thread environment. In order to accomplish this, a concept of a task is introduced in the PARRT software. The task is a block of pixels of final image and, instead of calculating pixel illuminations in a large double loop (over pixels rows and columns of image as a whole), the image is divided into number of the non-overlapping rectangles (each rectangle corresponding to a task) and the rendering is accomplished rectangle by rectangle.

When the POSIX threads enabled version of render compiled, a configurable number of threads is created upon the render launched. Each thread is then accessing the task queue to pick next rectangle of pixels for rendering. During rendering, threads are writing calculated pixels colors into final image residing in the shared memory. When completing the rendering of the current task, a thread is again accessing task queue and taking next rectangle of pixels to calculate, if any available. The task queue is single data structure that has to be protected by the locking. The queue is locked when a thread is taking next task from it, and unlocked as soon as first available task removed from the queue and assigned to the thread. These operations are lasting for very short time, so there is no danger of having threads starving for next input and thus the synchronization between threads is not affecting the parallel performance.

The OpenMP API ([15]) is primarily based on the compiler directives that may be used to explicitly direct the shared memory parallelism. Thus, while the POSIX threads API is operating system dependent and requires a POSIX compliant operating system to run, the OpenMP API is compiler dependent and requires an OpenMP supporting compiler to compile. An OpenMP support is often implemented in terms of the POSIX threads, but this is not the requirement. In such a way, it could be stated that the OpenMP API is more portable than the POSIX threads API, but still there exist no much compilers supporting OpenMP, so still there is no clean winner between these two approaches for shared memory paradigm parallel programming and that was the reason for deciding to support both in the PARRT software.

OpenMP makes it possible to define a region in code that will be executed in parallel through `parallel` directive. Further, this directive could be used to specify which program variables will be shared and which will be private. In our case, the default was set for each program variable to be shared, except for the task queue. Following

OpenMP construct used in the PARRT software is `for` directive. This is a work-sharing construct that divides the execution of the enclosed code region among the members of the team that encounter it. Trough `schedule` clause of this directive one could describe how the iterations of the loop are divided among the parallel executions and the dynamic schedule is selected as most appropriate schedule type for the PARRT. The iterations of the loop naturally represents solving tasks from the task queue, so first thing to do in the loop is to pick next available task from queue. Like with the POSIX threads implementation, this is single place in code where the parallel execution should be synchronized and with the OpenMP API this is accomplished through `critical` directive. This directive is creating a short critical section protecting the queue integrity and accomplishing the synchronization.

Since the OpenMP implementation for compiler used (Intel C/C++ compiler) resides internally on the POSIX threads API for its implementation on each platform used for testing (Linux, Windows), we weren't able to discern any impact of the choice between the POSIX threads and the OpenMP APIs on the parallelization performance. On the other side, while both APIs are relatively simple to employ, it could be stated that OpenMP is certainly easier to use and thus could be recommended over the POSIX threads API in case supporting compiler provided.

5. RESULTS

In order to present the results of the parallelization procedure, some measures have to be defined.

Definition 1. *Suppose (i) n is a number of the execution threads and (ii) t_p is the total execution time of the parallel version of the algorithm. Then the cost c of the parallel execution is calculated as:*

$$c = n \cdot t_p \tag{1}$$

Definition 2. *Suppose (i) t_s is the total execution time of the sequential version of the algorithm and (ii) c is the cost of parallel execution. Then the efficiency e of the parallelization is calculated as:*

$$e = \frac{t_s}{c} \tag{2}$$

The efficiency is value in [0, 1] range and if it is close to 1, then it could be stated that the parallelization of given algorithm is meaningful.

Before presenting efficiency results, one should mention that no much difference is discerned in the algorithm performance regarding the efficiency scheme used. Both bounding volume hierarchy and voxel grid performed similarly in the sequential, as well as in both parallel versions of the algorithm.

The efficiency results will be presented for the Mount test scene. Table 1 shows results for the bounding volume hierarchy case, while Table 2 contains results for the voxel grid case.

Table 1: Efficiency values for Mount scene and bounding volume hierarchy case

	8x8	*16x16*	*32x32*	*64x64*
2 threads	0.892	0.910	0.921	0.925
4 threads	0.857	0.894	0.904	0.901
8 threads	0.660	0.794	0.864	0.814
16 threads	0.316	0.578	0.777	0.695

Table 2: Efficiency values for Mount scene and voxel grid case

	8x8	*16x16*	*32x32*	*64x64*
2 threads	0.953	0.928	0.964	0.965
4 threads	0.920	0.938	0.951	0.948
8 threads	0.600	0.775	0.880	0.844
16 threads	0.235	0.515	0.744	0.728

Along the columns, efficiency values for different task (pixel rectangle) size are presented. Along rows, the different number of threads efficiency values is presented. The number of threads is equal to the number of active processors on corresponding multiprocessor machines used for testing. Presented results are for POSIX threads parallel version. As mentioned above, available OpenMP implementation uses POSIX threads internally, thus giving the same results.

The same results are depicted on Figure 2 and Figure 3.

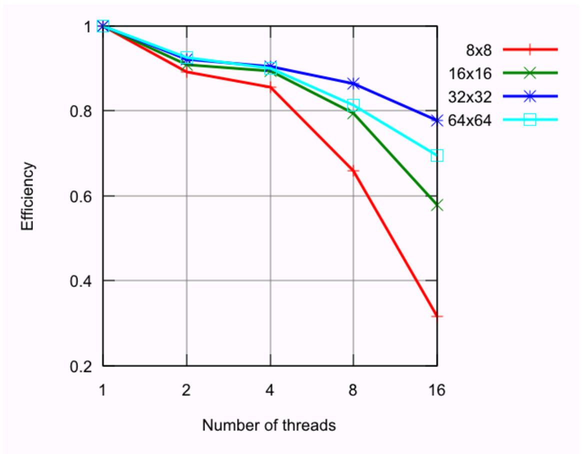

Figure 2: Efficiency values for Mount scene and bounding volume hierarchy case

From the results presented, it could be noticed that most of the time the efficiency values are above 0.8, and is considered as a very good result. On the other side, it is also noticeable that the efficiency values are rapidly dropping when the number of threads is above 8. This is caused by the fact that memory access becomes a bottleneck; namely, while there exist no memory conflicts between different threads (since all memory accesses are read operations), each thread is still accessing different parts of the scene and thus different areas in the memory. The memory could not serve all threads at the same time and for this reason the latency is introduced and the performance is decreasing.

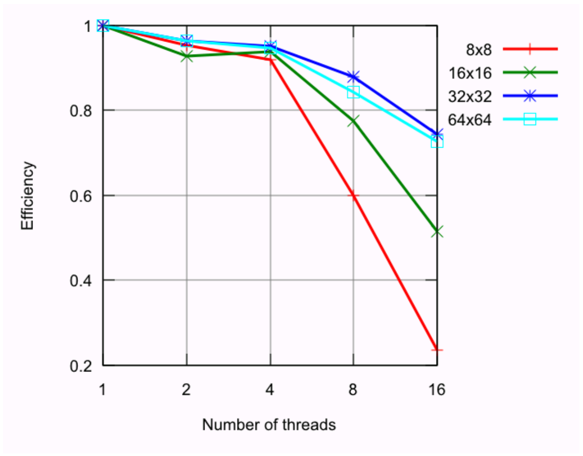

Figure 3: Efficiency values for Mount scene and voxel grid case

Another interesting result is that the best task size is 32x32 rectangles of pixels. This result pertains to 512x512 image size used for testing. Obviously, for lower task size the cache memory of each individual processor is not utilized most efficiently. Parts of the scene that given processor is accessing are stored in its cache memory and in case of bigger task size there is better locality and cached data are usable for longer time. When the task size is small, then this locality is not utilized well and the contents of given processor cache memory is replaced more often, thus increasing number of the shared memory accesses and decreasing overall performance. For example, we consistently measured, using Valgrind ([12]) memory usage analyzer, up to 15% more L1 and up to 10% more L2 cache misses each time when decreasing block size for given test scene on a 4-way system.

On the other side, when the task size too big, there always exist some processors that will have to do much more work on its last task than some other processors. Thus, at the end of the rendering some processors will be idle while some other processors still having lots of work to do. This will increase the total execution time and thus decrease the performance.

6. CONCLUSION

This paper examined an implementation of Ray Tracing algorithm for the multiprocessors. Two implementation variations are presented, one using the POSIX threads API and another one using the OpenMP API, thus covering both most used of existing methods for parallel programming on the shared memory parallel architecture.

After completing the implementation of sequential and then parallel version of the algorithm, a number of tests are conducted in order to measure the parallel version efficiency and examine the influence of changing working parameters (like task size) on the performance. The results are presented for a typical test scene, showing that parallel version achieves very good efficiency up to 8-way multiprocessor systems. For 16-way multiprocessor system, the efficiency is rapidly dropping and this is an expected result for a memory-access intensive application as Ray Tracing.

Still, the performance gains for 2-way, 4-way and 8-way multiprocessor systems are very significant. Since the parallelization procedure is relatively simple using both POSIX threads or OpenMP API once the sequential version of the algorithm structured properly, it could be certainly recommended approaching this procedure in case one need to render some scenes using Ray Tracing algorithm in as short time as possible and at the same time having access to the multiprocessor machine. We further think that our results are especially interesting in the context of new explicit multi-threading microprocessor developments ([17]), like Intel HyperThreading architecture.

An interesting direction for further research would be to parallelize PARRT software for the distributed model of parallel execution. While it could be expected that the parallelization procedure is more complicated than for the shared memory parallel execution case, it would be very interesting to compare the results that could be achieved in a distributed environment with results presented for a shared memory environment.

Acknowledgment: Authors are grateful to paper reviewers for very useful suggestions regarding various improvements over initial version.

REFERENCES

[1] Amanatides, J., and Woo, A., "A fast Voxel traversal algorithm for ray tracing", *Proceedings of Eurographics 1987*, 3-9.
[2] Booch, G., *Object-Oriented Design and Analysis with Applications*, Addison-Wesley, Reading, MA, 1994.
[3] Bouatouch, K., and Priol, T., "Parallel space tracing: An experience on an iPSC hypercube", *New Trends in Computer Graphics*, 1988 170-188.
[4] Bui-Tuong, P., "Illumination for computer generated pictures", *Communications of the ACM*, 6 (1975) 311-317.
[5] Crow, F.C., "3D image synthesis on the connection machine", *Parallel Processing for Computer Vision and Display*, 1 (1988).
[6] Gardener, R., "An internet-based distributed rendering system", personal communication, 2004.
[7] Goldsmith, J., and Salmon, J., "Automatic creation of object hierarchies for ray tracing", *Computer Graphics & Applications*, 5 (1987) 14-20.
[8] Haines, E., "A proposal for standard graphics environment", *Computer Graphics & Applications*, 11 (1987) 3-5.
[9] Jevans, D.A.J., "A review of multi-computer ray tracing", *Ray Tracing News*, 1 (1989) 8-15.
[10] Kay, T.L., and Kajiya, J.T., "Ray tracing complex scenes", *Proceedings of SIGGRAPH 1986,* in *Computer Graphics*, 4 (1986) 269-278.
[11] Lee, T.Y., Raghavendra, C.S., and Nicholas, J.B., "Parallel implementation of a ray tracing algorithm for distributed memory parallel computers", *Concurency – Practice and Experience,* 9 (1997) 947-965.
[12] Nethercote, N., and Seward, J., "Valgrind: A program supervision framework", *Runtime Verification Workshop*, 2003.
[13] Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I., and Omura, K., "LINKS-1: A parallel pipelined multimicrocomputer system for image creation", *Proceedings of 10th IEEE Annual International Symposium on Computer Architecture*, 1983.

[14] "POSIX 1003.1c standard", *IEEE,* 1995.
[15] Quinn, M.J., "Parallel programming in C with MPI and OpenMP", *McGraw-Hill*, 2003
[16] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., "Object-oriented modeling and design", *Prentice Hall*, *Englewood Cliffs*, *NJ*, 1991.
[17] Ungerer, T., Robič, B. and Šilc, J., "A survey of processors with explicit multithreading", *ACM Computing Surveys*, 1(2003) 29-63.
[18] Whitted, T., "An improved illumination model for shaded display", *Communications of the ACM*, 6 (1980) 343-349.