Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 6295-6302 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

TWO ALTERNATIVE PARALLEL IMPLEMENTATIONS FOR RAY TRACING: OPENMP AND MPI

Alexandre S. Nery[a], Nadia Nedjah[b] and Felipe M.G. França[a]

[a]*LAM – Computer Architecture and Microeletronics Laboratory, Systems Engineering and Computer Science Program, COPPE, Universidade Federal do Rio de Janeiro, {alexandre.solon@gmail.com, felipe@cos.ufrj.br}, http://www.coppe.ufrj.br~felipe*

[b]*Departamento de Engenharia de Sistemas e Computação, Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, nadia@eng.uerj.br, http://www.eng.uerj.br~nadia*

Keywords: 3D rendering, ray tracing, uniform grid.

Abstract. In computer graphics, rendering scenes into high-quality images efficiently is critical, especially if some interactivity is required. However, it is hard to satisfy both speed and interactivity requirements for most existing rendering algorithms. Ray Tracing is an algorithm for three-dimensional scene rendering, but it needs massive heavy floating-point computations. Nevertheless, the algorithm can be very well parallelized, making it ideal for parallel mutli-core architectures. In this paper, we present a structured model for parallel Ray Tracing together with two software implementations in OpenMP and MPI. The model is based on the *Uniform Grid* spatial subdivision of the scene, which allows for intersection test reduction. Furthermore, we show that a parallel implementation improves the algorithm performance when the rendered scenes are large and sparse.

1 INTRODUCTION

Rather than a simple rendering algorithm, Ray Tracing is a powerful simulation of light interaction within a three dimensional scene. Such interaction is determined through costly intersection computations between each light ray and the scene. Each intersection test may generate more rays and lead to even more computation, increasing the level of details in the synthesized image, which includes: shadows, transparency and mirror effects.

Tracing the path of each light ray is a computing intensive task. For a naive ray tracing implementation, in which every ray is intersected against the whole scene, the rendering time is proportional to the number of objects. Furthermore, acceleration data structures based on spatial subdivision of the scene can greatly reduce the number of intersection tests, since only the objects that are in the direction of a given ray are tested for intersection. Even thought, a frame rate above 60 fps (frames per second) is usually unachievable in ray tracing, which makes the algorithm unsuitable for real time rendering.

The algorithm is well-known for its parallelism, as every ray may be processed independently. Generally, a parallel ray tracer scales almost linearly to the number of available processors Cameron (2007). Thus, parallel architectures have been enabling real-time performance in Ray Tracing, such as multi-core processors Hurley (2005), GPGPU Purcell et al. (2005)(General-Purpose Graphics Processing Unit) and custom parallel designs in FPGA. In the latter, it has been possible to render scenes between 20 and 60 fps, at frequencies up to 90 MHz Woop et al. (2005).

In this paper, we present a parallel ray tracing model, along with two parallel algorithms based on that model. The architecture, called GridRT Nery et al. (2009), has been developed in FPGA, while the algorithm has been developed in OpenMP and MPI. The model exploits the *Uniform Grid* Fujimoto et al. (1988) spatial subdivision of the scene to reduce the amount of intersection tests as well as performing them in advance, as described in Section 3.

The rest of this paper is organized as follows: in Section 2, we briefly introduce the ray tracing sequential algorithm. Then, in Section 3, we present the parallel model. Thereafter, in Section 4, we sketch two parallel algorithms, the corresponding implementations and yielded results. Finally, in Section 5, we draw some conclusions and point out some directions for future work.

2 THE RAY TRACING ALGORITHM

The Ray Tracing algorithm is briefly introduced in this section, while more details are best described in textbooks on the subjectSuffern (2007). Thus, given a *Virtual Camera* pointed towards the scene, the idea is to launch a view ray (a.k.a. *primary ray*) that passes through a pixel of the *viewplane*, according to Fig. 1.

Beyond the viewplane, lies the scene to be rendered. Therefore, intersection tests are performed against each ray and, if and intersection is found, a *secondary ray* may be generated heading towards a new direction. Also, at each intersection point, the intersected object properties are stored in order to merge the results into a single pixel color. This idea is described in Algorithm 1, also known as *Whitted-Style Ray Tracing* Whitted (1980). Intersection and

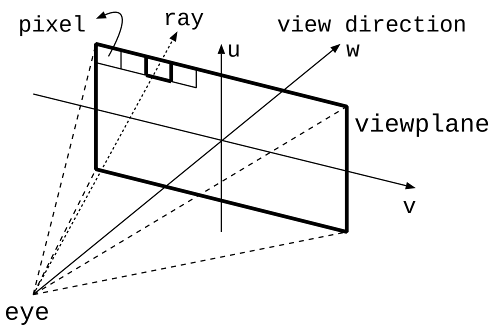


Figure 1: The virtual camera setup

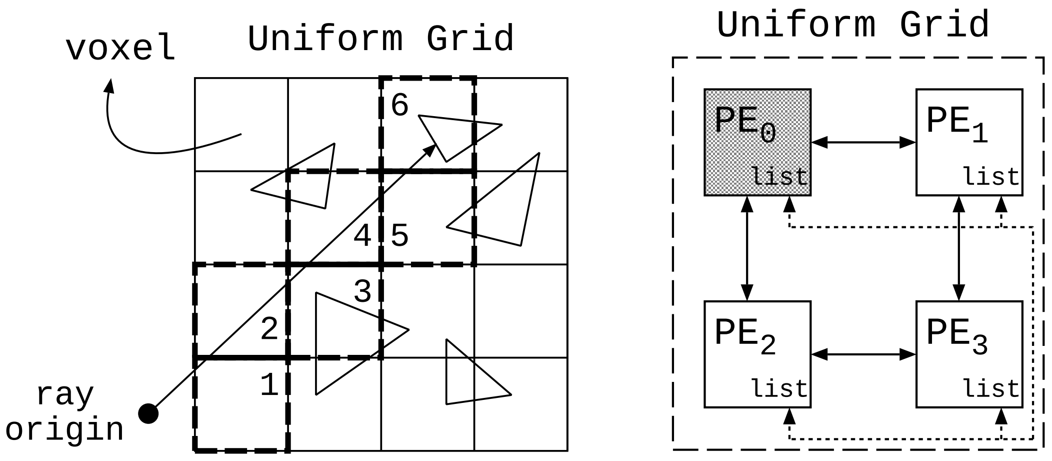
shading computations are are an essential part of the algorithm Shirley (2000); Phong (1998).

Algorithm 1: Simplified Ray Tracing Algorithm (*Whitted-Style*)

```
    Input: scene, ray, depth
    Output: pixel color
 1  if depth $>$ max then
 2      return black ;                  /* usually the background color */
 3  else
 4      lowest := $\infty$;
 5      foreach scene object do
 6          t := intersect(ray, scene[obj]);
 7          if t $<$ lowest and t $>$ 0 then
 8              lowest := t ;                    /* update the lowest value */
 9              result := scene[obj] ;                /* stores the object */
10      if result $\neq$ null then
11          if result.material is specular or transparent then
12              ray := o + t$\cdot$ d ;              /* compute new ray direction */
13              color := shade(t, result) ;          /* compute pixel color */
14              return color + trace(ray, scene, depth+1)$\cdot$result.frac
15          else
16              color := shade(t, result) ;          /* compute pixel color */
17              return color
```

At first, the algorithm checks the recursion depth to determine its conclusion. Such depth controls how further secondary rays may be spawned from an initial primary ray, which directly affects the level of transparency and mirroring details in the synthesized image. After that, intersection tests determine the closest intersection point to the ray origin and stores the intersected object and its properties. Then, if the intersected object material is specular or transparent, the algorithm is called recursively with a new ray direction. Otherwise, the object properties are used to determine the intersection color contribution. Back from recursion, those contributions are merged into a single pixel color.

Objects are usually represented as a collection of triangles, called *Triangle Mesh*. The ray-triangle intersection function, line 1, first determines the intersection between the ray and the triangle plane, as every triangle must lie in a plane. Then, if positive for intersection, the algorithm checks if such point belongs inside the triangle's coordinates. All the computation is accomplished through barycentric coordinates Shirley (2000).



(a) Traversal order (b) Grid Ray Tracing

Figure 2: The example of ray traversal order and the Grid Ray Tracing parallel model, in which the highlighted PE is master.

The shading function, lines 1 and 1, computes a color based on the position of each light source in the scene. Otherwise, every object would have a flatness appearance in the final image. Usually the color is computed as a combination of three components: ambient, diffuse and specular. Such shading model is known as *Phong Empirical Shading Model*Phong (1998), after Bui Tuong Phong.

3 THE PROPOSED MODEL

The parallel model is based on the Uniform Grid acceleration data structure, in which the scene is subdivided into regular regions, called *voxels*. Each voxel holds a piece of scene data and, hence, reduces the number of intersection tests. That is only those voxels that are pierced by a given ray have its scene tested for intersection, as depicted in Fig. 2(a). Those tests are performed in the order that grid is traversed, so the intersection point is guaranteed to be the closest to the ray origin.

Furthermore, our model maps each voxel onto a Processing Element (PE), which becomes responsible for computing intersection tests within its piece of scene data Nery et al. (2009), stored in a associated memory. So, every PE that is traversed by a ray is activated to process it in parallel, performing intersection tests in advance throughout the ray. However, it must decide which PE holds the correct result (the closest to the ray origin) at the end of the computation.

Thus, instead of exchanging the results between the activated processing elements, we still use the traversal order to determine the correct result. Thus, each PE is connected to its neighborhood by interrupt lines (Fig. 2(b)). So, according to the traversal order, once an intersection is discovered, an interrupt request is sent to the next PE in the traversal list, which is forwarded until reaching the last PE in the list. Moreover, a second type of interruption handles the situation when a PE requires a feedback from a previous one, before assuming that it has found the closest intersection point. For instance, let the traversal list be $L = (2, 0, 1)$ and that the second processing element has discovered an intersection. Then, PE_1 is interrupted, but PE_0 must wait a feedback signal from PE_2. Note that there is almost no overhead involved in handling interrupt requests, since interruptions are being handled in simultaneously by a separate process in every PE rather than by coded-interrupt routines.

Algorithm 2: Ray Tracing OpenMP Algorithm

```
   Input: scene, ray
   Output: pixel color
 1 numcells := grid.getNumCells();
 2 omp_set_num_threads(numcells) ;                        /* set #threads */
 3 shared_res[numcells] ;             /* resultant object of each thread */
 4 foreach ray of the viewplane do
 5     traverse := grid.traverseGrid(ray) ;            /* list of traversal */
 6     #pragma omp parallel private(tid, result)
 7     begin
 8         tid := omp_get_thread_num() ;                       /* thread id */
 9         triangles := grid.getTriangleArray(tid);         /* thread objs.  */
10         foreach triangle do
11             result := intersect(ray,triangle);
12             if $result \neq null$ then
13                 shared_res[tid] := result;
14             i := 0;
15             while $traverse[i] \neq tid$ do
16                 if shared_res[traverse[i]] $\neq$ null then
17                     shared_res[tid] := null;
18                     break ;             /* actually break innermost for */
19                 i := i+1;
20     end
21     Continue to shading computation;
```

4 THE PARALLEL ALGORITHM

Both algorithms presented in this section follow the model from Section 3, with minor modifications. In the MPI version, interrupt signals are implemented as messages while int the OpenMP version the communication is based on shared resources. In essence, the parallel computation is still the same, but the synchronization between processes is different, especially in the latter case.

4.1 OpenMP implementation

The OpenMP Ray Tracer maps every processing element onto a *Thread* and their communication is based on shared resources, as in Algorithm 2. First of all, a ray is traversed through the grid and a list containing the order of traversal is generated, line 2. Then, the parallel section begins with each thread getting the corresponding scene and intersecting it. However, the whole scene is not intersected at once. After each intersection test, the algorithm reads a shared array which stores the result computed so far by each thread (line 2). Thus, according to the order of traversal, if there is a result already stored by a previous thread, the current one aborts. At the end, only the correct result remains in the shared array.

4.2 MPI implementation

The MPI Ray tracer maps each processing element onto a *Process* and models interrupt signals as messages, Algorithm 3. A master process is in charge of distributing the scene data to

Algorithm 3: Ray Tracing MPI Algorithm

```
Input: scene,ray
Output: pixel color
if rank = 0 then
    Distribute necessary data to Processes;
foreach ray do
    Traverse the ray through the grid structure;
    if the Process rank is listed then
        foreach triangle do
            result := intersect(triangle);
            if $result \neq null$ then
                Determine the Process position in the traversal list;
                if first then
                    Send interrupt message to the next Process;
                if last then
                    Receive message from the previous Process;
                    Check the message and interrupt if necessary;
                else
                    Receive message from the previous Process;
                    Forward the message to the next Process;
                    Check the message and interrupt if necessary;
        if $result \neq null$ then
            Continue to shading computation;
    MPI_Barrier(MPI_COMM_WORLD) ;   /* Synchronization barrier */
```

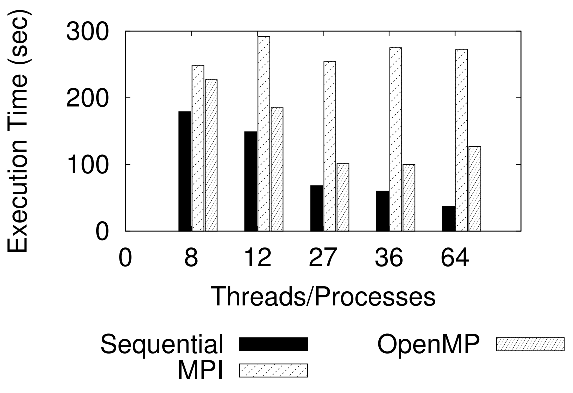
others (line 3). Then, for each ray-triangle intersection test, every process checks their position in the traversal list and decides whether to expect an message or/and to send an interrupt message (line 3).

4.3 Results

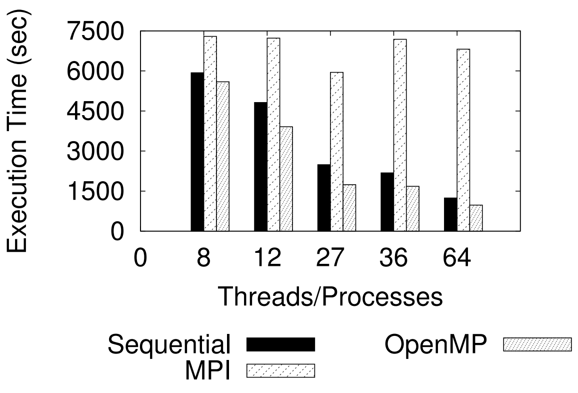
In this section, we present some performance results from the parallel algorithm execution, in Fig. 3(a) and Fig. 3(b), for primary rays. Such results were obtained in a *Core i7* 2.26GHz Intel architecture, which is capable of running eight threads in parallel. At each execution, the grid size configuration was increased: 8, 12, 27, 36 and 64 processing elements, which corresponds to the number of threads/processes.

Also, note that eight parallel threads are enough, since the number of threads actually doing some real work in parallel are those listed in the traversal list. All the others will be waiting in a *busy wait* style. Thus, for the grid configuration that we have setup, the maximum parallel threads/processes is eight, which fits nicely even for a 64 processors configuration.

The result depicted in Fig. 3(a) compare the sequential and parallel implementations, for a small scene of approximately 67K triangles. In that case, the sequential algorithm beats the parallel one, due to a synchronization overhead, especially for a large grid configuration. However, as the scene grows sparsely, the parallel OpenMP algorithm becomes a better alternative, as in Fig. 3(b). What happens is that the chance of occurring intersection tests in advance (in parallel) increases for distributed objects (Fig. 4(b)), while the sequential algorithm would step into each voxel looking for further intersections along the ray. On the other hand, the parallel



(a) 67K triangles.



(b) 1.20M triangles.

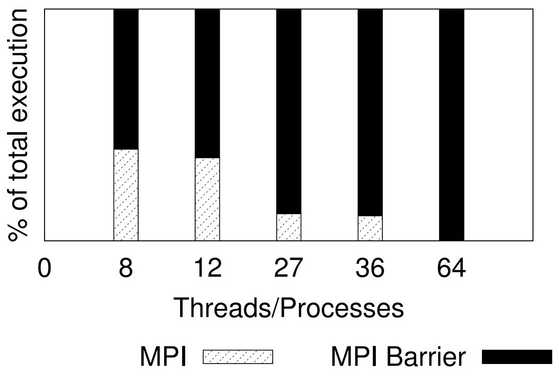
Figure 3: Parallel vs. Sequential Ray Tracing.

MPI algorithm did not performed as expected: most of its execution time is being spent on barriers, as depicted in Fig. 4(a). The execution time for both implementations are depicted in Table 1. We have not included the latency of MPI messages as they are almost inexistent, since all processes are running on the same machine.

Table 1: Execution times* for rendering 67K and 1.2M scenes.

Grid	67K scene			1.2M scene		
	Sequential	MPI	OpenMP	Sequential	MPI	OpenMP
8	179	248	227	5931	5675	5594
12	149	292	185	4819	4625	3909
27	68	254	101	2489	2010	1740
36	60	275	100	2182	1945	1678
64	37	272	127	1242	1200	975

*All times are in seconds.



(a) MPI barrier overhead

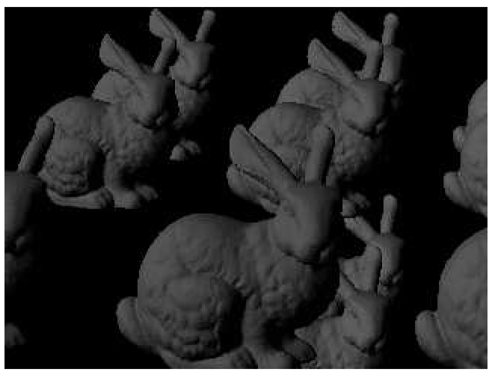

(b) 18 *Stanford Bunnies*

Figure 4: MPI Barrier overhead and 18 Stanford Bunnies render.

5 CONCLUSION AND FUTURE WORK

The GridRT architecture was presented together with two parallel algorithm solutions. Despite the lower performance achieved for small scenes, the parallel algorithm becomes promising as the scene size increases. Generally, computer graphics applications are required to render large scenes. Also, the MPI implementation is being re-structured in order to avoid the usage of barriers and so increase its performance. Furthermore, rays that are traversed through different processors could be processed in parallel, which is likely to increase both algorithms performance.

We are now working on the hardware implementation, with four processing elements only, due to area constraints. However, a hardware solution is expected to yield much better performance than the software parallel solution.

REFERENCES

Cameron C. Using fpgas to supplement ray-tracing computations on the cray xd-1. *HPCMP Users Group Conference*, 0:359–363, 2007. doi:http://doi.ieeecomputersociety.org/10.1109/HPCMP-UGC.2007.79.

Fujimoto A., Tanaka T., and Iwata K. Arts: accelerated ray-tracing system. *Tutorial: computer graphics; image synthesis*, pages 148–159, 1988.

Hurley J. Ray tracing goes mainstream. *Intel Technology Journal*, 9(2):99–107, 2005. ISSN 1535-766X. doi:http://dx.doi.org/.

Nery A.S., Nedjah N., and França F.M.G. Gridrt: A massively parallel architecture for ray-tracing using uniform grids. In *DSD '09: Euromicro Conference on Digital System Design*, pages 211–216. IEEE Computer Society, Los Alamitos, CA, USA, 2009.

Phong B.T. Illumination for computer generated pictures. *Seminal graphics: poineering efforts that shaped the field*, pages 95–101, 1998. doi:http://doi.acm.org/10.1145/280811.280980.

Purcell T.J., Buck I., Mark W.R., and Hanrahan P. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 268. ACM, New York, NY, USA, 2005. doi:http://doi.acm.org/10.1145/1198555.1198798.

Shirley P. *Realistic ray tracing*, pages 35–38. A. K. Peters, Ltd., Natick, MA, USA, 2000. ISBN 1-56881-110-1.

Suffern K. *Ray Tracing from the Ground Up*. A. K. Peters, Ltd., Natick, MA, USA, 2007. ISBN 1568812728.

Whitted T. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/358876.358882.

Woop S., Schmittler J., and Slusallek P. Rpu: a programmable ray processing unit for realtime ray tracing. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 434–444. ACM, New York, NY, USA, 2005. doi:http://doi.acm.org/10.1145/1186822.1073211.