Nested Parallelism Concepts of Ray Tracing Algorithms and Multithreading API Performance Analysis

Aleksandar Ristovski, Marjan Gusev, Sasko Ristov

***Abstract*—Since the breakthrough of backward ray tracing algorithms, computational grids have proved to be a necessity in the development of the high-quality graphic industry. They can provide infrastructure for nested and multileveled parallelism in addition to high computing power, hence, meeting both the demanding requirements and nature of complex ray tracing. In addition, multithreading techniques, such as OpenMP and Pthreads, can further optimize the rendering by taking use of the multi-core architecture present at the processing end-units.**

***Index Terms*—Ray Tracing;Multilevel Parallelism;Supercomputing;Multithreading;OpenMp; Pthreads;**

I. Introduction

Ray tracers and computational grids share close connection as the commercial convenience of these algorithms requires tremendous computing power. Such examples make the rendering farms owned by Pixar Studios, Marvel, Bad Robot, Disney and others, that produce either visual effects or entire 3D animated films, which also happen to be some of the most powerful supercomputers of the present time [1].

The research conducted consists of two main parts:

- Analysis of the ray tracing algorithm in order to do mapping of its steps to a hierarchy of supercomputing entities for efficiency improvement
- Series of test cases performed by the designed ray tracing program, that examine the efficiency of multithreading APIs, such as OpenMP and Pthreads, with additional resolution of the tests' results

In order to support the concept of the former, a brief introduction of the principles of ray tracing is given, explaining the algorithm in sufficient details beneficial to creating the connection to a multileveled parallelism. Furthermore, the latter supplements as a logical extension of the notion, covering multithreading within a single process that executes on a multi core end-unit, including an exhaustive performance analysis.

The rest of the paper is organized as follows. Section II gives an overview of the ray tracing process. Decomposition of the algorithm and mappings of the algorithm steps to different levels of parallelism are presented in Section III. An approach of one way of algorithm mapping and its results are described in Section IV. Section V discusses the relevant conclusions and future work.

II. The Ray Tracing Algorithm

The program that incorporates a ray tracing algorithm is typically called a **ray tracer**. Such ray tracer simulates a model of visual perception the light to object interaction. Under those terms, the ray tracer aims to recreate the way any optical sensor would perceive the environment, following the laws of physics and the means of analytical geometry equations for the purpose.

The pioneer ray tracers were designed in a way that they virtually recreated the path light travels from light sources, bounces off of surfaces and disperses into the surrounding. Thus, the picture is created with the color information carried out by the light rays that manage to reach the virtual screen, whose content is displayed as an outcome of the whole process. Evidently, this is a very expensive method of simulation, referring to all those unnecessary calculations: the light rays that end up apart from the screen. What made ray tracing algorithms plausible is the ray backtracking, i.e. **backward ray tracing**[2]. Instead of firing rays from light sources, rays are being fired from the computer screen. When the ray intersects an object, the color information for the intersection point is calculated. The number of calculations is therefore significantly reduced. However, the rendering of complex scenes still detains high computing power.

The objects in the scene are commonly known as **primitives**. They can represent spheres, planes and above all, polygonal meshes. The triangular and quadrangular meshes are the most prevalent meshes among the meshes made up of other polygonal forms because they deliver most accurate 3D models of real life objects. Each polygon of the mesh structure is treated as individual primitive.

For rendering results that can best resemble realistic scenes, it is necessary to incorporate shadows, reflection and refraction as part of the ray tracer[3]. That is why additional rays are fired from the initial intersection point. In general terms, the rays can be divided into four different types, as shown in Fig. 1, each with its own purpose, properties and role in deriving the definite color of a pixel. **Tracing rays**, or primary rays, are the ones that are fired from each of the screen pixels and in the end of the process, bring the final color information.**Shadow rays** are used to determine whether an object will or will not get illumination color from a light source. **Reflection rays** are fired in case the primitive intersected with the tracing ray has reflective properties. **Refraction rays** are fired in case the primitive intersected with the tracing ray is partly or completely transparent.

Primitives' illumination can be done by either local or global **illumination model**[4]. One of the most efficient local

978-1-5090-4086-5/16/$31.00 ©2016 IEEE

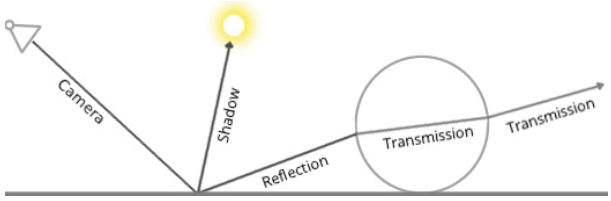

Fig. 1. Types of rays used for ray tracing

illumination models is the **Phong Illumination Model** (PIM). That is the illumination model used in the ray tracer designed for the testing. It incorporates three elements of illumination: ambient, diffuse and specular component. These elements are respectively responsible for resembling omnipresent light, three-dimensional object properties and the specular reflections due to glossy surfaces. The concept of **anti aliasing** (AA) has its purpose for smoothening the distortion artifacts that appear when rendering different color border areas. For a single pixel, several tracing rays, often called AA rays, are being fired, with origin points and directions defined by a respective AA algorithm. Afterwards, the colors of the AA rays are mixed to define the final color of the primary ray. The grid AA algorithm is used for optimal qualit[5].

III. Mapping The Steps of a Ray Tracing Algorithm

It is common knowledge that every algorithm whose steps are performed over a set of data is followed by the possibility for its parallel execution. Furthermore, ingrained cycles give the opportunity for nested parallelism. The goal was to map the steps of the ray tracing algorithm to a supercomputing approach that can best exploit the benefits of operating in parallel. There are a few conventional techniques that implement parallelization into ray tracing, mainly including KD tree traversals for object culling, diffuse inter-reflections, as well as other shading enhancements [6][7]. The algorithm mapping discussed is of different nature because it takes into consideration the infrastructure of the computational resources. The infrastructure is typically a distributed rendering farm[8][9], that incorporates RPC-based programming models. The temporally dissembled algorithm consists of the following phases:

- Divide the image screen into distinctive pixels;
- Fire AA tracing rays for each of the screen pixels;
- Perform individual intersection tests with the corresponding objects in the scene, for each of the AA rays, to determine the object that intersects first with the tracing ray;
- Calculate color for the intersection point according to PIM;
  - The diffuse and specular components are calculated individually for each light source;
  - Shadow rays are fired towards each light source, and again, intersection tests are performed with corresponding objects present in the scene except for the winning object (since an object cannot be shadowed by itself), to determine whether light is gathered from a particular light source;
- Calculate the color derived from reflection and refraction rays;
- Mix the colors supplied by PIM, reflection and refraction, by taking into account the reflection and refraction properties respectively, in furtherance to setting the final color of the AA pixel;
- Mix the colors from the AA rays, to set the final color of a an actual pixel;

A number of this steps can be executed in parallel, by:

- Split the work by dividing the screen (the picture) into several subscreens (sets of pixels) that are to be rendered independently;
- Split the work for each pixel by rendering the AA rays independently;
- Split the work for calculating the final color of an AA ray, regarding to PIM, reflection and refraction component individually;
- Split the work for calculating the specular and diffuse components of PIM, by doing the calculations for each light source independently, while firing shadow rays along;

Naturally, it is no particular challenge to take into practice a single method of parallelization. It is way more challenging to incorporate several of them. The most forward approach regarding the algorithm mapping is to split the pixels among CPU units of computational grid. Hence, after the pixels are done rendering, the complete picture is assembled. Although it may seem that a preferable option is to have one CPU per pixel, that is seldom attainable, since it is inefficient to constantly adjust the number of CPUs according to the number of pixels for rendering, or have redundant number of CPUs. Moreover, it is not the number of pixels that is crucial, but the number of AA subpixels. An idea that can take advantage of the computational grids without making too many inconsistencies between the number of pixels and the deployment of computational resources is to generate a definite set of AA rays pending to be fired. This centralized approach emphasizes homogeneous distribution of the AA rays to the processors of the computational grid. After the AA rays finish their work, a set of computational units is in charge of figuring out the pixel colors by taking average value of the corresponding AA pixels. A genuine image is assembled a while later by putting the individual pixels together. This concept utilizes the stream programming model[10].

A very significant aspect of supercomputing in general is maintaining the equilibrium of load balance. How much work will have to be done for a single AA pixel is something that cannot be predicted, as each AA pixel has a lifespan of its own, with its neighbor pixels' work being absolutely irrelevant to it. Hence, if we distribute an assignment of a single AA pixel to each of the processors, the processing time will have no coherence, in response to the unpredictable amount of work for an AA pixel. That is why is better to keep the processors

busy after they are done with their work. For that purpose, we can choose to have a pool of AA pixels waiting to be rendered. That way, after a processor is done rendering an AA pixel, it can be assigned a new one. Hence, suitable custom build kernel is required for the comprehension.

It is due to incoherence of the pixel work that the ray tracing algorithm is unstable for the GP GPUs architecture. However, the development of GPUs in recent years has made them plausible for certain tasks that are part of the ray tracing process, such as shading techniques. Thus, GPUs can only accompany CPUs, but cannot replace them at the rendering farms[11][12].

Furthermore, the multicore architecture can be used for grained approach on CPU level. The mechanisms for reflection, refraction, and the individual diffuse and specular components regarding the respective light sources can be processed individually, hence, run on separate threads. The reflection and refraction rays are likely to span threads themselves, which is why these threads should not be synchronized. Similar to the case of potential load imbalance when rendering AA pixels, the cores of a processor can be left idle when some threads finish before others. When a thread finishes its work it should join the main thread, freeing up its processor resources to other pending threads. In a scene with numerous light sources and objects that have reflective and/or refractive properties, the number of threads can reach thousands. For optimal performance, only a single thread should completely occupy a core until its task is done, since threads' context switching is redundant and will only give the system unnecessary overhead. That way, the system will have optimal load balance on both CPU and core level.

IV. Multithreading API Performance Analysis

Choosing an API or standard that will handle the multithreading is of high importance to the speedup factor. Different APIs/standards have individual syntax for thread initialization, argument passing and termination. In addition, they handle the thread management distinctively. OpenMP and Pthread (standard POSIX.1c, package pthreads-w32) were used for multithreaded implementation of the algorithm explained in III.

Since a desktop workstation machine was used for the testing, not all of the previously mentioned concepts could have been implemented as they require custom built process and thread scheduler, i.e. a custom kernel. Instead, the pixels were fragmented according to the number of threads and each thread was given its own pixel pool fragment for rendering. For each test case, a different parameter set was used. Thus, the parameters that varied regarding their values were the following: number of point light sources present in the scene: 1, 2, number of spheres present in the scene: 1, 2, 3, 4, number of running threads 1, 2, 3, 4, 5, 6, 7, 8, number of AA pixels per pixel: 1, 4, 9, recursion depth: 0, 1, 2. All together, there were 576 different test cases. Each test case was executed 5 times in a row to insure little aberration. Afterwards, the average time measured was taken into consideration and was written into the tests' log file for further analysis. Each execution of the test case was done sequentially, in parallel via OpenMP and in parallel via Pthreads. Hence, a scene was rendered 8640 times in total. The system was using a central processing unit Intel Core i7-3610QM that has 4 cores which support multithreading, work frequency of 2.3 GHz, max turbo frequency of 3.3GHz and 6MB of L3 cache, as well as 2 x 4GB DDR3 memory units, with clock rate of 1600MHz. The tests were run on Windows 7 64bit platform.

For each test case, the time needed for rendering in parallel via OpenMP and via Pthreads was measured, as well as the time needed for serial rendering. The speedup and efficiency of both approaches was additionally calculated. Equation 1 was used for calculating the speedup, while Equation 2 for calculating the efficiency. Furthermore, the ratio of OpenMP and Pthreads execution times was used as an indicator of which did better. One of the best indicators of how much work was done by the ray tracer is the number of intersection tests that occur, as intersection tests are the mechanism that incorporates reckoning most of the key parameters for the upcoming color mixing. Therefore, it is important to include the number of intersections in the log file as well.

$$S(n) = \frac{\text{execution time using one thread}}{\text{execution time using n threads}} = \frac{t_s}{t_p} \quad (1)$$

$$E(n) = \frac{\text{execution time using one thread}}{\text{execution time using n threads} * n} = \frac{t_s}{t_p * n} \quad (2)$$

In 10% of the test cases, OpenMP did worse than serial In 10% of the test cases, OpenMP did worse than serial execution, with no close connection to the setting of a specific parameter. This was due to the big overhead OpenMP can give the system. The same thing happened for Pthreads as well, but in half the number of cases it did for OpenMP, i.e. 5%.

Super speedup is rare occurrence that happens when data fits the processor cache in a way that it saves data fetching between memory and multiple levels of cache, eventually saving time that would not have been spared during serial execution. Although such cases are unusual, corresponding ones were reproduced while rendering with 5 threads and having 5912376 intersection tests. That opens up the opportunity to recreate such scenarios even though the total number of intersections does not match the number of intersections of the super speedup occurrence, i.e. if the number of intersections is lower than expected, we can compensate by performing suitable number of redundant operations that will not affect the final outcome. This technique is almost regular practice for achieving super speedup for a number of algorithms.

OpenMP had worst speedup when rendering with 3, 4 and 5 threads, while the speedup factor of PThreads had almost linear growth and reciprocal decrease for fixed recursion depth and fixed AA depth, regarding the number of threads. It is worth saying both the speedup and efficiency were slightly better when using higher AA depth. A local maximum of the minimum speedup is present when rendering with 6 threads.

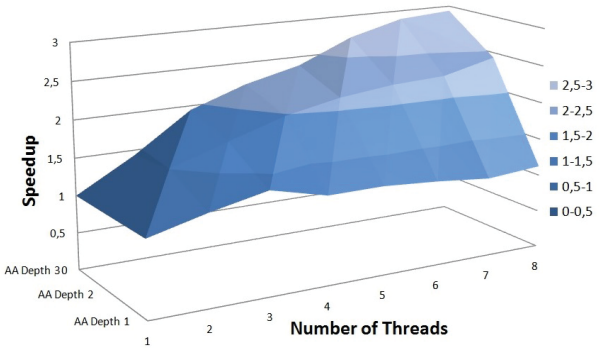

Fig. 2. Maximum speedup values for OpenMP: 2 light sources, 2 spheres, recursion depth 2

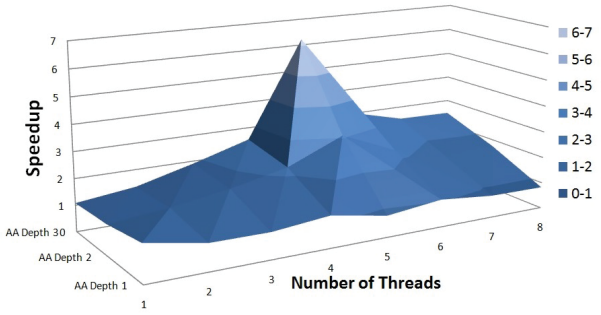

Fig. 3. Maximum speedup values for Pthreads, super speedup: 1 light source, 1 sphere, recursion depth 2

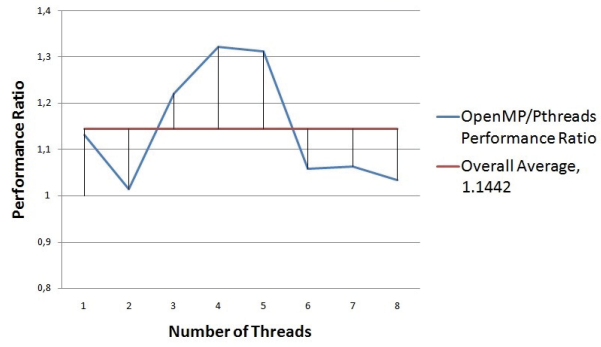

Fig. 4. Overall performance comparison

Number of Lights	1	1	1
Number of Spheres	1	1	1
Number of Threads	**5**	**5**	**5**
Antialiasing Depth	3	3	3
Recursion Depth	2	0	1
Number of Intersections	5912376	5912376	5912376
Speed Up: Serial vs. PThreads	**6,21914**	**6,09481**	**5,97867**
Efficiency: Serial vs. PThreads	124,38%	121,90%	119,57%

Fig. 5. Super speedup cases when rendering with Pthreads

The speedup increase is almost of logarithmic nature, with noticeable hump when rendering with PThreads, using 5 and 6 threads, Fig. 2, Fig. 3 . The hump evolves into super speedup for certain combination of the scene parameters. The super speedup cases are itemized in Fig. 4. The information from Fig. 5 best represent the overall performance, as it show the average ratio of the rendering time.

V. Summary and Conclusions

Graphic studios turn to ray tracing more often as no other method can render graphics of such high quality. That is why huge effort is being made in designing appropriate infrastructure that can handle the rendering of complex scenes, while on the other hand, experts in ray tracing need to devote themselves in composing suitable algorithm mapping to the computational resources. The efficiency is assorted by the load balance attained both among the CPUs and the CPUs' cores. Hence, small grained approach is preferred and is administered by handling the AA pixels and the various aspects an object can get its color throughout independent tasks.

The charts presented in the second part of the research pragmatically reflect the performance of the ray tracer when executed with various numbers of threads. Such analysis is fundamental when the ray tracer succumbs to commercial use, in order to obtain the maximum from the computational resources. According to analysis of this type, the system design should take into consideration the choice of processor architecture, so that the system can best cope with the work it is given.

References

[1] J. Křivánek, M. Fajardo, P. H. Christensen, E. Tabellion, M. Bunnell, D. Larsson, A. Kaplanyan, B. Levy, and R. Zhang, "Global illumination across industries," *SIGGRAPH Courses*, 2010.

[2] J. Arvo and M. Chelmsford, "Backward ray tracing," in *Developments in Ray Tracing, Computer Graphics, Proc. of ACM SIGGRAPH 86 Course Notes*, 1986, pp. 259–263.

[3] J. T. Kajiya, "The rendering equation," in *ACM Siggraph Computer Graphics*, vol. 20, no. 4. ACM, 1986, pp. 143–150.

[4] P. Shirley and R. K. Morley, *Realistic ray tracing*. AK Peters, Ltd., 2008.

[5] M. A. Dippé and E. H. Wold, "Antialiasing through stochastic sampling," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 69–78, 1985.

[6] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o (n log n)," in *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006, pp. 61–69.

[7] M. Levoy, "Efficient ray tracing of volume data," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 3, pp. 245–261, 1990.

[8] J. Bigler, A. Stephens, and S. G. Parker, "Design for parallel interactive ray tracing systems," in *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006, pp. 187–196.

[9] J. Yao, Z. Pan, and H. Zhang, "A distributed render farm system for animation production," in *International Conference on Entertainment Computing*. Springer, 2009, pp. 264–269.

[10] J. Gummaraju and M. Rosenblum, "Stream programming on general-purpose processors," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 2005, pp. 12–pp.

[11] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. CRC Press, 2008.

[12] D. E. Wexler, L. I. Gritz, E. B. Enderton, and C. W. Everitt, "Load balancing," Jan. 11 2011, uS Patent 7,868,891.