Parallel Ray Tracing using MPI and OpenMP

S. M. Ashraful Kadir[1] and Tazrian Khan[2]

[1] Scientific Computing, Royal Institute of Technology (KTH), Stockholm, Sweden
smakadir@csc.kth.se,
WWW home page: http://www.csc.kth.se/~smakadir/
[2] ICT Entrepreneurship, Royal Institute of Technology (KTH), Stockholm, Sweden
tazrian@kth.se

Abstract. In this paper we present an implementation of parallel ray-tracing in both distributed memory and shared memory systems using MPI and OpenMP respectively. We have developed an object oriented solution using C++ and performed experiments in both the platforms. We present the performance model analysis with memory and time complexity analyses. Experimental results show that the implemented static load balancing technique achieves linear speedup and near optimal efficiency in both the cases. We also present a comparison among different load-balance scheduling schemes in OpenMP.

Keywords: Ray tracing, high performance computing, parallel computing, MPI, OpenMP, load balancing, parallel speedup, parallel efficiency

1 Introduction

Ray tracing is a technique of developing an image from a 3D scene by tracing the trajectories of light rays through pixels in a view plane. This technique is well known for its capacity of producing a very high degree of photorealism, which is a clear advantage over the industry standard raster graphics techniques. Although ray tracing is popular for its quality to generate photo-realistic images, for its expensive computational cost it is only applied where the image can be rendered slowly ahead of time. The computer graphics animation in the modern films is an example of such industries. In industries like computer games, where real-time image rendering is a precondition, ray tracing remained prohibitive and the raster graphics techniques were dominating. Possibilities of applications of ray tracing in streamline industries where real-time or fast image generation is necessary were faint until recently. In the recent years the advancement of high performance computing and combinatorial algorithms enabled processing of large amount of computational tasks in a much smaller time. As a consequence ray tracing became potential to be applicable in interactive visualizations.

Ray tracing is a so called embarrassingly parallelizable algorithm and it is naturally implemented in multi-core shared memory systems or distributed systems. In this article we present the implementation of a parallel ray tracer using MPI and OpenMP with static load balancing. Experimental results show that

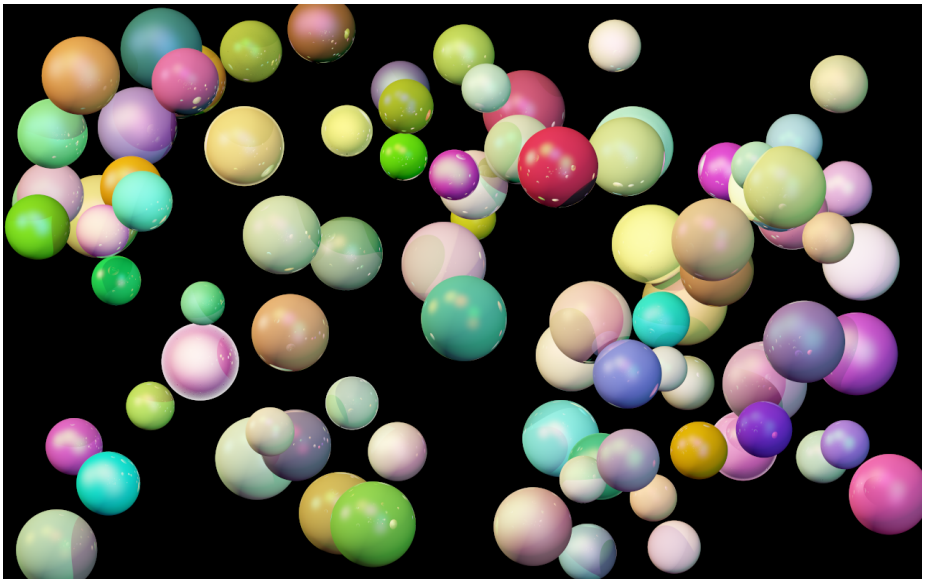

Fig. 1. Image created by scene configuration 1 (the scene contains 100 sphere objects, five point light sources, image created with 256 super-sampling coefficient, 648.007 seconds time required to render the scene using MPI with 32 processors)

our implementation achieved near linear speedup. The experimental results also show that the implemented load balancing algorithms are effective for both the distributed memory and shared memory systems to achieve near optimal efficiency.

1.1 Motivation

In the recent years increased use of multi-core shared memory and distributed systems, availability of increased computing performance and cost reduction in high performance hardware make the ray-tracing potential to be used in the industry applications. The topic received overwhelming researches and developments interest in the high performance computing (HPC) and computer graphics industries. Hurley's article "Ray Tracing Goes Mainstream" in Intel's technical journal in 2005, c.f. [4], advocates that the performance of ray-tracing reached the stage where it is feasible that it will take over from raster graphics in a near future for interactive gaming and other applications. As a consequence of such discussions, in 2007 [12], Shirley et. al. predicted, based on their expertise in computer graphics and computer architecture, that interactive ray tracing would become more prominent in the near future. This is a very active research field in the current world.

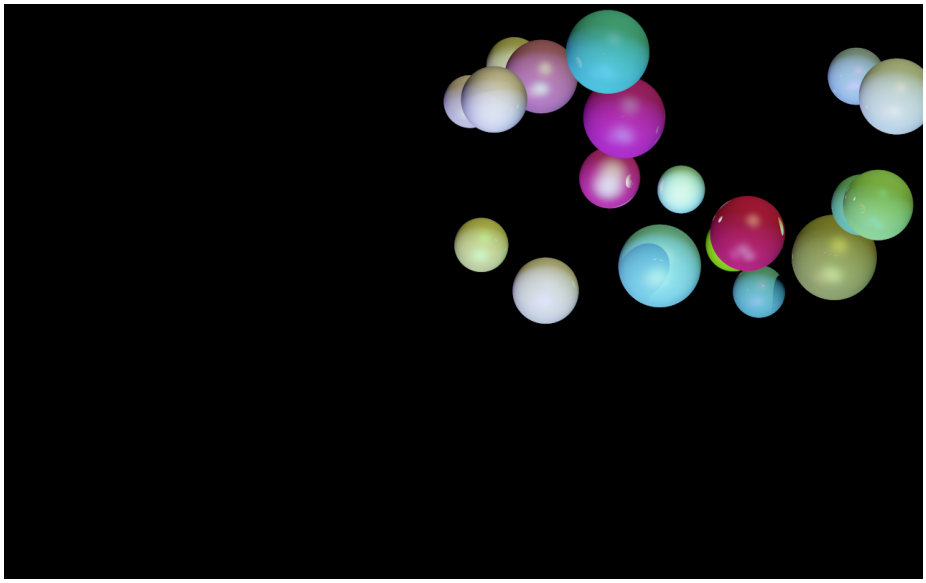

Fig. 2. Image created by scene configuration 2 (the scene contains 20 sphere objects, five point light sources, image created with 256 super-sampling coefficient, 58.1082 seconds time required to render the scene using MPI with 32 processors)

Although many researchers in this field predict that ray-tracing will gradually take over the computer graphics industries over industry standard raster graphics algorithms, there are many complex issues still need to be settled. Hardware support, optimal utilization of resources, system efficiency, parallel computing platform, optimal algorithm etc. are the parameters that need to be understood and resolved to bring ray-tracing in mainstream industrial applications. In this circumstances, we intended to implement and experiment with a simple parallel ray-tracer using the industry standard MPI and OpenMP for distributed systems and multi-core shared memory systems respectively.

1.2 Related Works

Although the computational expensive nature kept ray tracing less-interesting for real-time and interactive visualization and for applications where fast image generation is necessary, many researchers and practitioners developed algorithms and systems for ray tracing. Specially ray tracing techniques became attractive to many ray tracing enthusiasts and they employed hours to days of computer processing power to produce a single image. Examples are available in "the Internet Raytracing Competition", an internet based competition of ray traced photo-realistic image generation [13].

Ray tracing was first developed in 1968 by Appel and at the same time by Goldstein and Nagel, c.f. [1]. Appel was the first to ray trace shadows, whereas Whitted and Kay extended ray tracing to handle specular reflection and refraction. Since then ray tracing is in interest of the researchers in computer graphics fields. A comprehensive discussions and background of ray tracing is available in [1].

Parallelization of ray tracing algorithms is in active research interest for over the last one decade. Reinhard and Jansen presented an implementation of a parallel ray tracer with data parallel and hybrid (demand driven) scheduling approach [10]. They used the PVM library to implement the distributed ray tracer. They claimed to achieve a good load balance by combining demand driven and data driven tasks. Freisleben et. al. presented a case study of several parallel versions of POV-ray raytracing package using MPI on workstation clusters and introduced an adaptive load balancing scheme where larger tasks are automatically assigned to faster processors [2]. Aleksandar et. al. implemented raytracing in multiprocessors using POSIX threads and OpenMP [11]. They achieved very good efficiency up to 8 way multiprocessor systems. Plachetka presented a demand-driven parallelization of raytracing based on POV-Ray and PVM and claimed the technique as perfect load balancing algorithm for image space subdivision [8]. The technique uses distributed object database and thus specially effective for very large image rendering that do not fit into single processor's memory. Johansson et. al. presented a parallel ray tracing system based on OpenMPI and LAM/MPI [7]. For load balancing the master assign tasks to the processing nodes as first-come-first-serve basis. The result is claimed to be close to optimal speedup considering the relative performance of each node. Heirich et. al. [3] applied a dynamic load balancing method based on diffusion model to a parallel Monte Carlo rendering system on parallel and cluster computers and claimed to achieve nearly optimal performance. Parallel interactive ray tracing system design on Manta software has been presented in [6] that uses reconfigurable components to achieve scalability on large number of processors where they proposed parallel pipeline model and achieved scalability on multiprocessor systems. A dynamic task scheduling algorithm for parallel progressive raytracing on distributed memory is proposed in [5]. Qureshi-Hatanaka [9] presented an empirical study of current load balancing system for heterogeneous distributed computing systems. They suggested an adaptive task assignment strategy and task migration from slow performance nodes to higher performance nodes in order to improve performance for run time task scheduling.

In this paper we present a simple parallel ray tracer, which is implemented for both MPI and OpenMP. We have achieved near optimal efficiency using simple static load-balancing technique.

2 Implementation

We have implemented an object oriented parallel ray tracer using C++ that can be compiled to run either sequentially or in parallel mode using MPI or OpenMP.

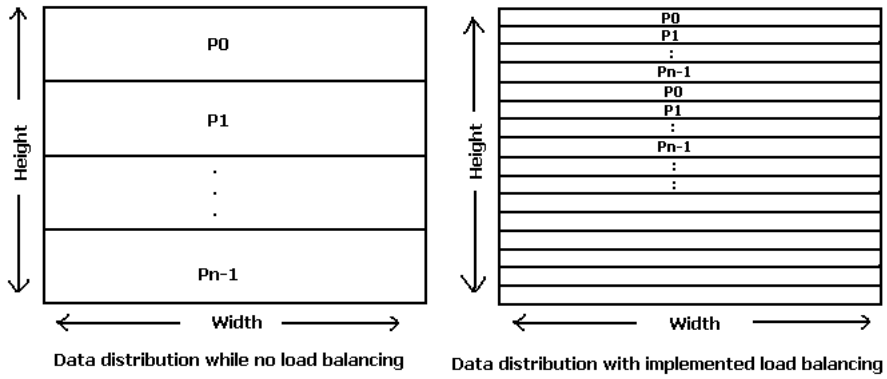

Fig. 3. Processors (or threads) wise work distribution

Load balancing schemes are implemented and also "load matrix" can be prepared for the viewport pixels which allows analysis of load distribution and optimal load balancing for certain scene. These features is decided on compile time based on the defined parameters. In the current implementation only sphere shaped objects are supported. However, new shape classes can be implemented. Random scene can be generated with given number of objects, light and boundaries of the 3D boxes where the objects and lights can be placed. The scene configurations can be saved in XML file format to be loaded at a later time to use the same configurations for experimentation. This allows to generate scene with different load distributions and can be effective for experimental purposes.

The program takes three parameters where first two are input config file name, image output file name. The last one is optional that take the configuration saving file name. The input file is read by all the processes (for MPI) or by the master thread (for OpenMP). The rendering work is done in parallel where each process (or thread) render only the segments allocated to them. The rendered image is collected to master process using recursive doubling method or the MPI collective communication function 'MPI_Gather' in case of running with MPI. The image file is saved in TGA file format.

The image scene is segmented among the processes (or threads) in similar size while running the program with no load balancing. The segmentation is done in height order. In case of using implemented load balancing, each process (or thread) starts processing the row with the same process (or thread) rank and incremented by number of the processes (or threads) until reach the image height. Figure 3 shows data distribution for both the cases. Three color values (R, G, B) are stored for each rendered pixels.

We have implemented load measurement options that can be enabled with compiled time parameter. Workload have been calculated with the numbers of floating point operations performed in each pixel and stored in the image size matrix. The calculated load computed by each processes is collected using re-

cursive doubling method or "MPI_Gather" while running the programme with MPI.

3 Performance Model

Complexity of ray tracing at a given pixel in the viewport is entirely scene dependent and in general an *a priori* load measurement is of non-deterministic nature. Ray tracing algorithm uses global illumination techniques where any small change in the scene can impact the rendering complexity of the pixel significantly. The implemented system measures the load per pixel of the viewport *a posteriori*. The load measurement works in single processor or in parallel run mode. Since this is a supporting system and not a integrated part of the ray tracer, we take the time performance measurement after disabling load measurement feature.

For an image of width X and height Y where the program is running on P processes (or threads) the memory and time complexity is given below.

3.1 Memory Complexity

The scene configuration is read by each processes (for MPI) or the master thread (for OpenMP) where the scene contains lights (6 float values) and objects (24 float values). So the memory requirement for scene configuration of the system is:

$$\text{For OpenMP:} \quad (24L + 96B) \quad \text{bytes}$$

$$\text{For MPI:} \quad (24L + 96B)P \quad \text{bytes}$$

where L is number of lights and B is number of objects.

For MPI, each process also has the memory requirement for image matrix where the matrix stores three RGB color (unsigned character) values for each rendered pixels. The average memory requirement (for MPI case) of P processors for the image matrix is $x = (3XY)/P$ bytes.

The overall memory requirement for image matrix for P processors (for the case of recursive doubling in MPI),

$$\begin{aligned}
& Px + \frac{P}{2}x + 2 \cdot \frac{P}{4}x + 4 \cdot \frac{P}{8}x + \cdots + \frac{P}{2} \cdot \frac{P}{P}x \quad \text{bytes} \\
&= Px + \underbrace{\frac{P}{2}x + \frac{P}{2}x + \cdots + \frac{P}{2}x}_{\log_2 P \quad \text{terms}} \quad \text{bytes} \\
&= Px + \frac{P}{2}(\log_2 P)x \quad \text{bytes} \\
&= Px(1 + \frac{1}{2}\log_2 P) \quad \text{bytes} \\
&= 3XY(1 + \frac{1}{2}\log_2 P) \quad \text{bytes}
\end{aligned}$$

For OpenMP, the memory requirement for the image matrix is $3XY$ bytes and for the collective communications in the case of MPI using 'MPI_Gather' the explicit memory requirement (in bytes) is

$$Px + (P-1)x = 3XY\left(1 + \frac{P-1}{P}\right) \approx 6XY.$$

So the overall memory requirements for the system is

$$\text{For OpenMP:} \quad (24L + 96B) + 3XY \quad \text{bytes}$$

$$\text{For MPI (P2P):} \quad (24L + 96B)P + 3XY\left(1 + \frac{1}{2}\log_2 P\right) \quad \text{bytes}$$

$$\text{For MPI (Collective):} \quad (24L + 96B)P + 6XY \quad \text{bytes}$$

Here we see that, for large P, the memory requirements for recursive doubling will be more than 'MPI_Gather'. For very large image (comparing to the memory requirements for the image matrix with the memory requirements for objects and lights) and large P, the memory requirements for recursive doubling is roughly $\log_2 P^{1/4}$ times than for 'MPI_Gather'.

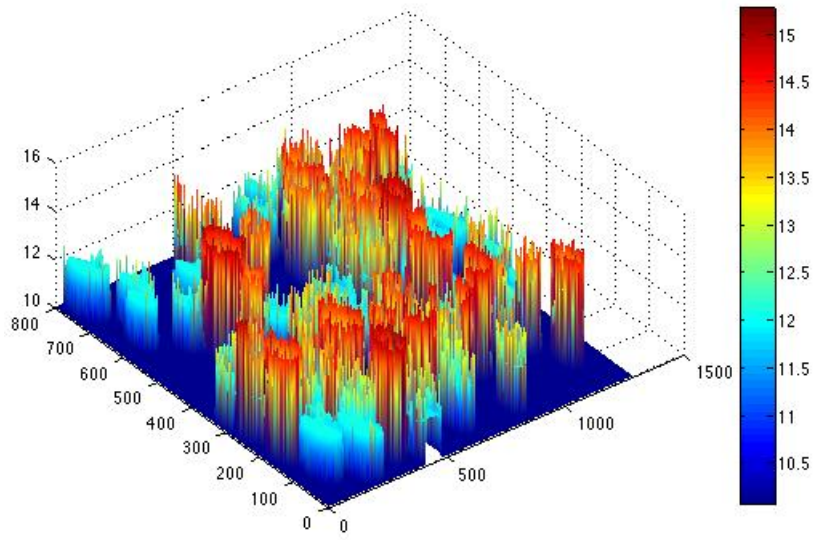

Fig. 4. Pixel wise load distribution for "scene 1" (load in logarithmic scale, image is rendered using super-sampling coefficient, $c = 8$), see figure 1 for the related image

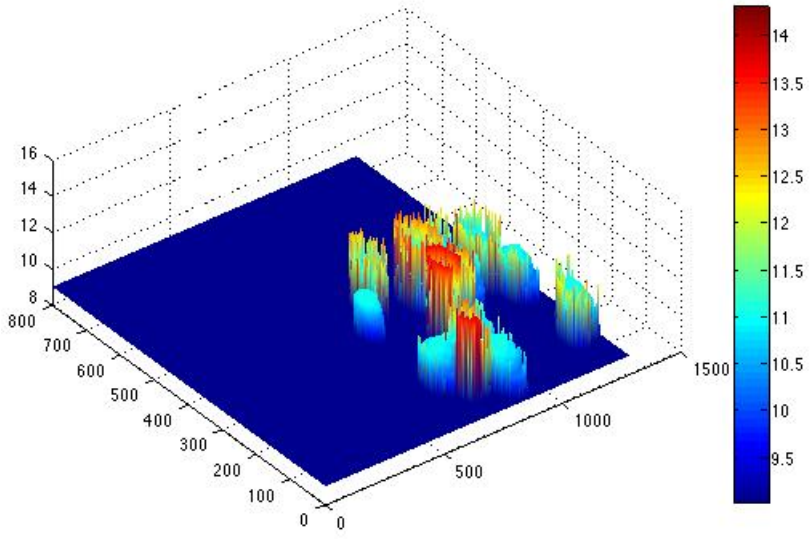

Fig. 5. Pixel wise load distribution for "scene 2" (load in logarithmic scale, image is rendered using super-sampling coefficient, $c = 8$), see figure 2 for the related image

3.2 Time Complexity

The computation time requirement for sequential programme is

$$T_s = \tau_f W \quad \text{seconds}$$

where τ_f is time for a floating point operation and W is the total work load defined as

$$W = \sum_{i=1}^{XY} (d(S) + w(S, x_i, y_i)),$$

where $d(S)$ is the fixed computation load required for each pixel dependent on the scene, S (e.g. in this implementation, a ray needs to check intersection with all objects in the scene) and $w(S, x_i, y_i)$ is the load which depends on the scene, S and the image pixel (x_i, y_i).

In average the computation time requirement for parallel program with optimal load balancing is

$$T_{comp} = \frac{\tau_f W}{P} \quad \text{seconds}$$

For the OpenMP case, we have no communication time requirement, since every thread access the shared memory and updates their own part of data. There can be some idle time when some threads complete their tasks and others

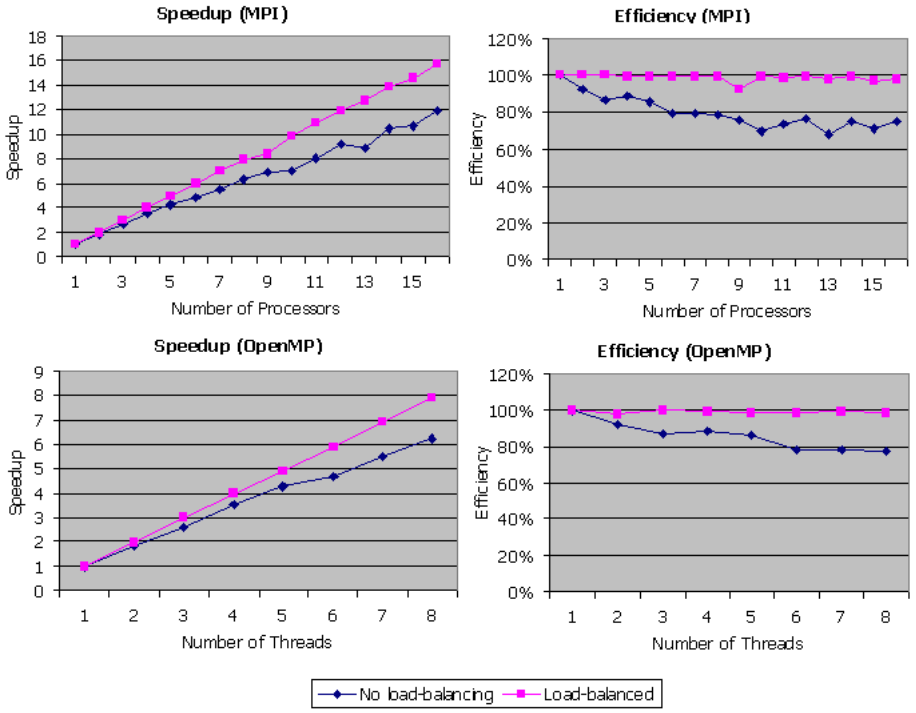

Fig. 6. Speedup and Efficiency for MPI and OpenMP (the execution time was taken for "scene 1" rendering with super-samping coefficient, $c = 16$)

are in running mode. We are interested to minimize the idle time T_{idle} to achieve optimal load balancing. For OpenMP case we have the total rendering time

$$T_{OpenMP} = T_{comp} + T_{idle} = \frac{\tau_f W}{P} + T_{idle} \quad \text{seconds}$$

Now the speedup for OpenMP case is

$$S_{OpenMP} = \frac{T_s}{T_{OpenMP}}$$

And corresponding efficiency is

$$\eta_{OpenMP} = \frac{S_{OpenMP}}{P}$$

For MPI case, we need to take the communications time in account. For ray-tracing communications are required to gather the image data to master processor from all other processors to print the image file. We have implemented

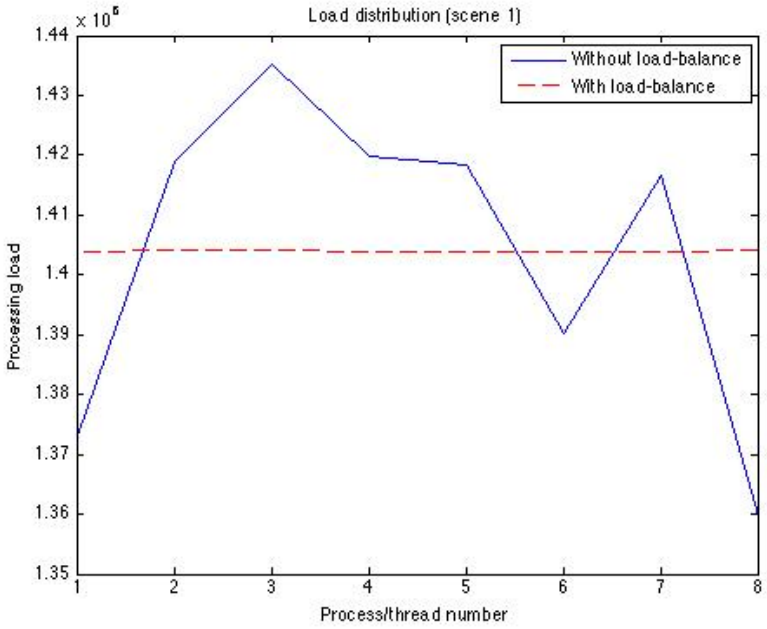

Fig. 7. Load distribution (in logarithmic scale) among 8 processors (or threads) for MPI or OpenMP parallel run for scene 1, see figure 4 for related pixel-wise load distribution

recursive doubling and also "MPI_Gather" to gather the data. We have the communication time

$$T_{comm} = \tau_s + \tau_b D \quad \text{seconds}$$

where τ_s is the latency or startup time, τ_b is the time to send one byte and D is the number of bytes to send. In our case for recursive doubling using peer to peer communications, the data transferred in parallel is

$$D = \frac{3XY}{P}\left[1 + 2 + 4 + \cdots + \frac{P}{2}\right] = \frac{3XY}{P}(P-1) \quad \text{bytes}$$

since for each pixel we store three unsigned characters for the RGB values. We performed an empirical comparison of time-complexity between recursive doubling and "MPI_Gather" that we will see in the result section. Again there can be idle time T_{idle} where one processor might be still working while others finished processing already. Now we have the time complexity for distributed systems,

$$T_{MPI} = T_{comp} + T_{comm} + T_{idle} = \frac{\tau_f W}{P} + \tau_s + \tau_b D + T_{idle} \quad \text{seconds}$$

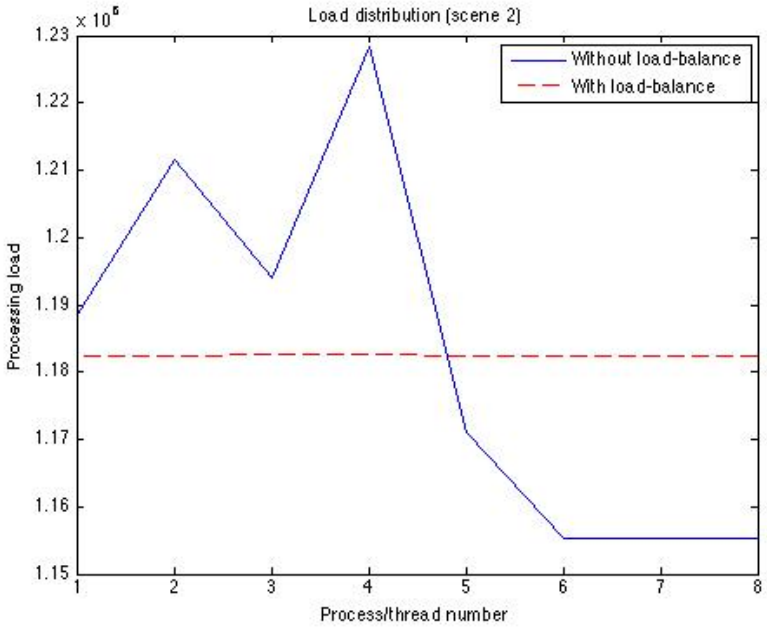

Fig. 8. Load distribution (in logarithmic scale) among 8 processors (or threads) for MPI or OpenMP parallel run for scene 2, see figure 5 for related pixel-wise load distribution

The speedup for MPI case is

$$S_{MPI} = \frac{T_s}{T_{MPI}}$$

And efficiency for MPI case is

$$\eta_{MPI} = \frac{S_{MPI}}{P}$$

4 Result and Discussions

For this experiment we have generated two random scenes with with different load distribution. The "scene 1" contains 100 sphere objects and the "scene 2" contains 20 sphere objects. Both the scenes have five point light sources. The scene 1 has a more widely distributed load across the pixels of the viewport where the scene 2 has major load at the upper-right quarter of the image. Figures 1 and 2 show the images corresponding the scene 1 and 2 respectively and figures 4 and 5 show the pixel wise load distribution where both the images were rendered with super-sampling coefficient, $c = 8$. The supersampling is an antialiasing

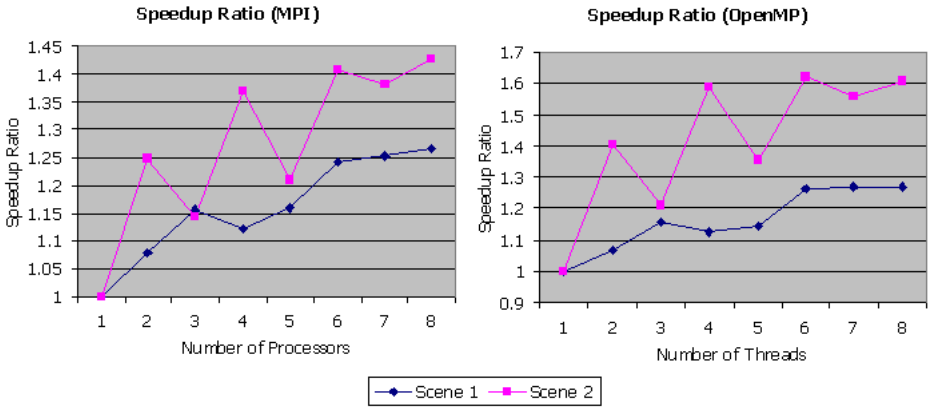

Fig. 9. Speedup ratio, $r_S = S_1/S_0$, where S_0 is the speedup achieved without any load balancing and S_1 is the speedup achieved with the static load balancing

technique used to add greater realism to a digital image by smoothing jagged edges specially on curved lines and diagonals or where the color gradient is higher. It is the approach of taking more than one sample for each pixel and combining them [1]. An example of the impact of super-sampling coefficient in the rendered images is available in figure 10.

It is important to note that the performance results from the distributed systems and multi-core systems used in this experiment are not directly comparable, since the hardware infrastructure of the two systems are not equivalent. For the experiments using MPI we have used the distributed memory computer system, namely Lenngren at Center of Parallel Computers (PDC), Royal Institute of Technology (KTH), Sweden, consisting of 442 PowerEdge 1850 servers where each node have two 3.4GHz "Nocona" Xeon processors and 8GB of main memory. The theoretical peak performance of the system is 6TFlop/s. A high performance Infiniband network from Mellanox is used for MPI traffic. On the other hand for experiments using OpenMP we have used the shared memory system, named Ferlin, where each node has two quad-core Intel Harpertown 2.66GHz CPUs (E5430) and 8Gbyte memory.

4.1 Compiler Performance with Optimization

We have experimented with GNU C++ compiler version 4.1.2 and Intel C++ compiler version 10.1 changing the optimization key. Table 1 in page 13 shows the result. Clearly Intel compiler performed better in all cases. Intel compiler performance is optimization key invariant for this case where the GNU C++ compiler performance without any compiler optimization is prohibitive.

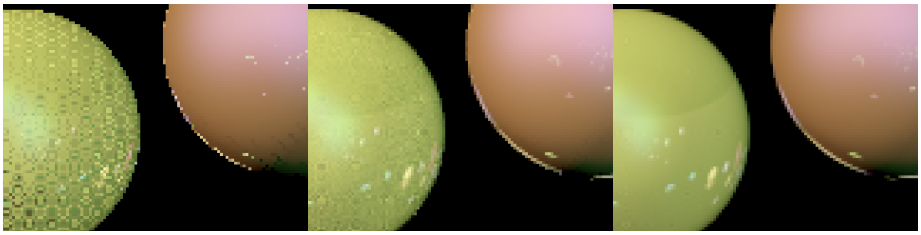

Fig. 10. Changes in image quality with different super-sampling coefficients, c. The three images from the left are rendered using $c = 1, 8, 256$. Although the difference between the middle and right image is not evident as with the left image, this is because the difference is clearer when we zoom more into the image. (The reader using electronic version of this document may zoom in the document more to verify this statement. The image quality difference is strong where the color gradient is higher.)

Table 1. Execution time for program compiled with different compilers and optimization key (the other parameters remained the same, the 1280×800 size image was rendered for a scene with 100 objects and 5 point light sources, super-sampling coefficient 16, using OpenMP with 8 threads)

Compiler	**Version**	**Optimization**	**Time (s)**
g++	4.1.2	-	201.03
g++	4.1.2	O2	16.70
g++	4.1.2	O3	16.30
icpc	10.1	-	12.70
icpc	10.1	O2	12.69
icpc	10.1	O3	12.71

4.2 Speedup and Efficiency

Ray-tracing algorithm is embarrassingly parallelizable and thus we expect linear speedup and optimal efficiency in parallel execution. The experimental result shows that we have achieved linear speedup and near optimal efficiency for both MPI and OpenMP cases using the static load-balancing technique implemented. Figure 6 shows the speedup and efficiency plots for MPI and OpenMP for the load-balanced and non-load-balanced cases.

4.3 Load-balancing

The simple static load-balancing technique we have implemented in this experiment achieved significant improvement in the performance. Figures 4 and 5 show the load distribution in logarithmic scale for the scene 1 and 2 under discussion. Processor- (or thread-) wise load distribution in a typical parallel run case for 8 processors (or threads) are given in figures 7 and 8 for scenes 1 and 2 respectively. Without any load balancing for both the scenes there were performance

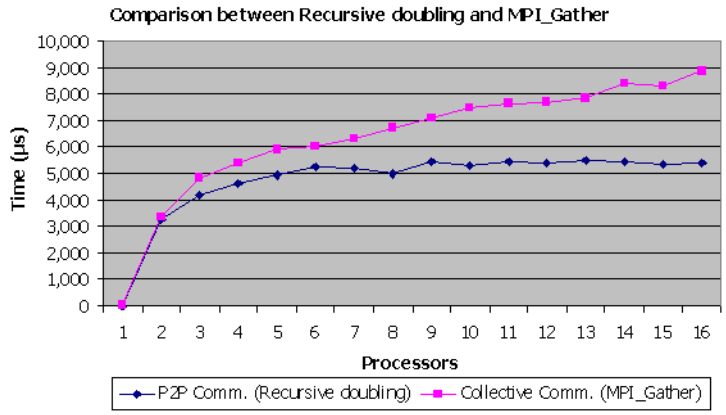

Fig. 11. Time-comparison between performance between recursive doubling (peer to peer communications) and MPI_Gather (collective communications). The time is computed for collection of a image of size 1280×800 where each pixel contains three unsigned characters for red, green and blue color intensities, i.e. the time requirement for $1280 \times 800 \times 3$ bytes total data collection.

bottlenecks due to unbalanced loads among the processors (or threads). By implementing static load balancing we achieved a better load distribution among the processors (or threads) as we see in figure 9. As expected, load-balancing improves the performance more in the case of scene 2, c.f. figure 2, where the load distribution is more uneven and concentrated completely in the top-right quarter pixels of the image.

4.4 Peer to Peer vs. Collective Communications in MPI

We have done a performance comparison between the implemented recursive doubling method based on peer to peer (P2P) communication among the processors and 'MPI_Gather' function based on collective communication. Figure 11 shows the performance graph for the time requirements in micro seconds. In this experiment the recursive doubling technique seems to perform better from the time requirements point of view. However, the communication time is very small in comparison to the computation time for this experiment and hence we did not find much impact with the changes of communication algorithm. Again, as saw in the memory complexity analysis, for large P, the memory requirements is less for 'MPI_Gather' than for recursive doubling. Also implementation of data communication using 'MPI_Gather' is much simpler than recursive doubling. Thus for this experiment, 'MPI_Gather' is a natural choice for such data collection.

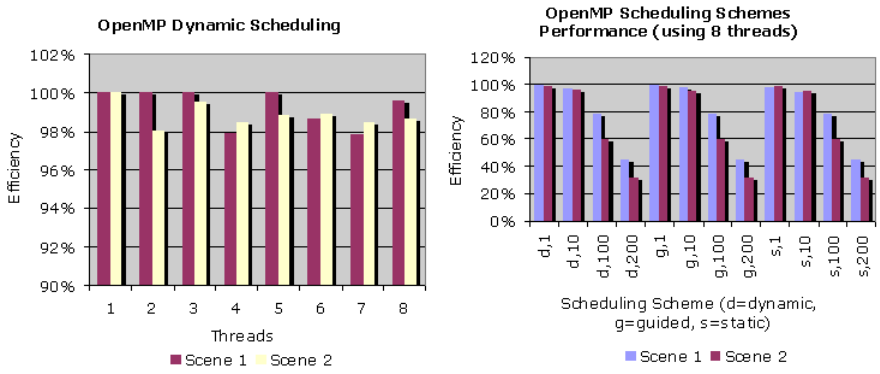

Fig. 12. Efficiency gained using "for loop parallelization" in OpenMP in runtime load balance scheduling. The left figure shows the efficiency gained using "dynamic" load balance scheduling for both scenes 1 and 2 and the right figure shows impact of different load balance scheduling.

4.5 Runtime Load-balance Scheduling in OpenMP

Figure 12 shows the experimental result for different load-balancing schemes in "for loop" parellelization using OpenMP. We implemented the "for loop" parallelization to decide the load-balance scheduling at runtime using the environment variable "OMP_SCHEDULE". However, the experimental result does not seem to be impressive as the unit work load that can be dispatched is on each row basis. The work distribution can be further partitioned per pixel basis which should allow improved load balancing that we will consider in future works.

4.6 Quality Variant Image Rendering

Image quality is a complex phenomena and would vary with a number of different parameters. In this experiment, we simplify the requirements and take the super-sampling coefficient as the only variable parameter. Since all other parameters remain constant, the image quality is proportional to the super-sampling coefficient. Clearly reducing the super-sampling coefficient produces less quality image as we see in the figure 10.

As a motivation of the quality variant image generation we may think of the video broadcasting over Internet industries. Apparently video over Internet is a popular entertainment media although the quality of broadcasted video over Internet is nowhere comparable with television media or so. For real-time ray tracing the quality of the image may need to be compromised in some cases, or in other words to be more general, it may be an easier approach to adjust the quality of the rendered image to achieve certain performance on particular hardware. This can be raised as an optimization problem and also can be taken as analogous to the accuracy-performance relationship in scientific computing applications.

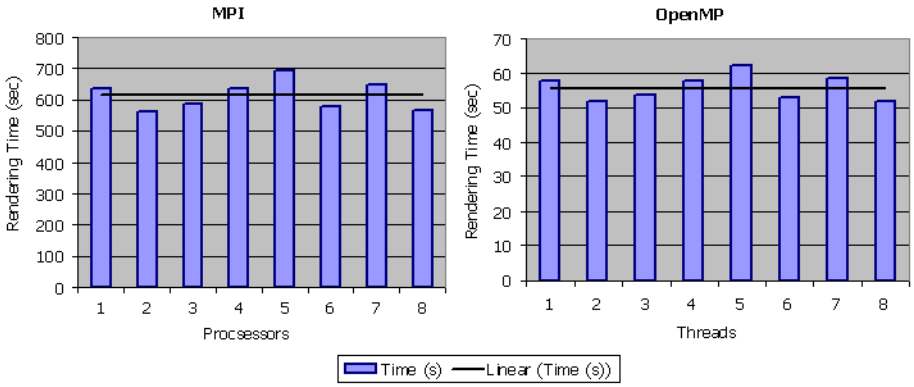

Fig. 13. Time required in parallel run for constant ratio $c/P = 8$, where c is the super-sampling coefficient and the number of processors (or threads), $P = 1, \ldots, 8$.

We have experimented with quality variant image rendering keeping the ratio between super-sampling coefficient, c and number of processors (or threads), P constant. The experiment shows that the rendering time required for different (c, P) pairs are banded and the the linear trend-line is horizontal for both MPI and OpenMP, as we see in figure 13.

5 Conclusion and Future Works

We have implemented an object oriented (using C++) parallel raytracer for both distributed memory (using MPI) and shared memory (using OpenMP) systems. Load measurement option can be enabled in compile time while the performance measurement has been done disabling the load calculation. Experimental results show that the implemented static load balancing technique achieves linear speedup and near optimal efficiency in both MPI and OpenMP cases where more enhanced performance achieved in uneven distribution of objects. Compilation performance has resulted better optimization in case of Intel C++ compiler version 10.1 than GNU C++ compiler version 4.1.2.

The implementation has some limitation in case of load balancing. The implemented load balancing might not be effective while running the programme in heterogeneous system where the computational nodes have different capacities. In such case a dynamic load balancing approach will be preferred, such as master/worker approach etc. Currently sphere objects have been implemented for rendering but the system design is open to incorporate additional objects categories by implementing new object classes. Another limitation is observed that the system still requires considerable rendering time where the objects positions are set outside of the image boundary. The future works are needed to implement dynamic load balancing and for the improvement of raytracing engine to support additional object shapes and optimization of ray tracing algorithms.

6 Acknowledgements

The work is a part of the course "Introduction to High Performance Computing" at Royal Institute of Technology (KTH), Sweden. The experiments were conducted using the high performance computing resources at Center for Parallel Computers (PDC), KTH. We acknowledge Dr. Michael Hanke, Associate Professor, CSC, KTH for valuable suggestions.

7 Download

Source code available in `http://code.google.com/p/simple-ray-tracing`.

References

1. Foley, J. D., van Dam, A., Feiner, S. K. and Hughes, J. F.: Computer Graphics Principles and Practice (second edition), Addison-Wesley Publishing Company, ISBN 0-201-12110-7 (1990)
2. Freisleben, B., Hartmann, D. and Kielmann, T.: Parallel raytracing: a case study on partitioning and scheduling on workstation clusters, Volume 1, Issue 7-10 (1997), Proceedings of the Thirtieth Hawaii International Conference on System Sciences
3. Heirich A. and Arvo J.: A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing, Volume 12, Numbers 1-2, pp. 57-68(12) (1998), The Journal of Supercomputing
4. Hurley, J.: Ray Tracing Goes Mainstream. Intel Technology Journal, Volume 9, Issue 2 (May, 2005) 99-107
5. Notkin I. and Gotsman C.: Parallel Progressive Ray-tracing, Volume 16, Number 1, pp. 43-55 (1997), The Eurographics Association 1997, Blackwell Publishers
6. Bigler J., Stephens A. and Parker S. G.: Design for Parallel Interactive Ray Tracing Systems, page(s): 187-196, IEEE Symposium on Interactive Ray Tracing 2006 (2006)
7. Johansson G., Nilsson O., Sderstrm A. and Museth K.: Distributed Ray Tracing In An Open Source Environment, Proceedings of the SIGRAD 2006 Conference (2006), Linkping University Electronic Press
8. Plachetka, T.: Perfect Load Balancing (Demand Driven) Parallel Ray Tracing, ALCOMFT-TR-02-95, Alcom-FT Technical Report Series (1995)
9. Qureshi, K. and Hatanaka, M.: An introduction to load balancing for parallel raytracing on HDC systems, Current Science, Vol.78, No.7, pp.818-820 (2000)
10. Reinhard, E. and Jansen, F. W.: Rendering large scenes using parallel ray tracing, Volume 23, Issue 7 (1997), Elsevier Science Publishers B. V. Amsterdam, The Netherlands
11. Samardzie, A. B., Starcevic, D. and Tuba, M.: An implementation of ray tracing algorithm for the multiprocessor machines, Volume 16, Issue 1 (2006), Yugoslav Journal of Operations Research
12. Shirley, P., Sung, K., Brunvand, E., Davis, A., Parker, S. and Boulos, S.: Rethinking Graphics and Gaming Courses Because of Fast Ray Tracing, International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH educators program (2007)
13. The Internet Raytracing Competition, WWW link: `http://www.irtc.org/` - last access date: October, 2008