Journal of Computational Innovations and Engineering Applications 1(2) 2017: 22–32

Efficient Load Balancing Technique for Parallel Ray Tracing Using a Reservoir

Joy Alinda Madamba and Francis Joseph Seriña

***Abstract*—Imitating the real world in emulations, such as the effect of light, is an important research area in software applications. However, a computing machine cannot precisely imitate all the aspects of light in short processing periods. Ray tracing is a technique that renders and accurately depicts light with a high time delay. Parallelizing the computers that run the rendering is not enough to reduce its running time. To further increase computing efficiency and improve ray tracing's applicability for mainstream purposes, load balancing must be performed. In this study, a proposed alternative load balancing technique that uses a reservoir was implemented to tackle the issue of using heterogeneous computers in the network and to minimize the communication overhead introduced by parallel applications. Results show that when using a reservoir, high speed-up is achieved at 80% initial task distribution while high efficiency is achieved between 20–75% initial task distribution. With speed-up as the priority, reservoir achieves at most 87% better speed-up than the scatter decomposition and 36% than the hybrid technique. It also has at most 28% better efficiency than the scatter decomposition. Finally, the reservoir algorithm achieved at most 25% better achieved speed-up (ASU) than the scatter decomposition and 10% than the hybrid.**

***Index Terms*—Computer Graphics, Load Balancing, Parallel Programming, Ray Tracing, Rendering**

I. Introduction

THE recent trend in computer development has been towards shorter execution times for complex applications. However, these applications have developed faster than the advances done in computer architectures. Single-core processors have thus given way to multi-processors, paving the way for parallel solutions to these complex problems. An example of this complex application is ray tracing. Many people consider ray tracing to be the best image synthesizing technique to date [1]. Ray tracing naturally shows precise reflections, shadows, and transmissions by following the path of light and applying the laws of geometric optics. Unfortunately, high quality images can only be produced at a large amount of time. Ray tracing has not been used in mainstream graphics yet, aside from the film industry, because no computing machine can single-handedly render large images in a relatively small useful amount of time [2]. Thus, ray tracing is a good candidate for implementation in a network of computers that would execute that application in parallel.

The use of parallelization has its own set of areas that need improvement when it comes to efficiency. One such issue is load balancing. The workloads assigned to different computers are not always equal and a balancing scheme must be applied in such a way that the computers complete their task at the same time [3]. Load balancing techniques are classified as static and dynamic. Static load balancing distributes tasks before processing, taking into account certain assumptions such as tasks with equal workload unless otherwise stated. Static load balancing techniques are based on the partitioning of the image plane (screen). One static technique found to be effective for network systems with less than 128 computers is Scatter Decomposition [4], [5]. In this algorithm, the tasks are alternately distributed to increase the probability that machines get equal workload amounts. It distributes the pixels in an alternating sequence (i mod p) where the p is the number of computers and i is the pixel to be rendered. The result of the alternating sequence formula is the computer, which renders that pixel. For example, Figure 1 shows how scatter decomposition is applied unto a picture with 100 pixels per row. Given that there are 3 computers within the network, the 1st, 4th, 103rd, 106th, etc. pixel is assigned to the first computer while the 2nd, 5th, 101st, 104th, etc pixel is given to the second computer and finally, the 3rd, 6th, 102nd, 105th, etc. pixel is given to the third computer.

J. A. Madamba is with the Electrical and Electronics Engineering Insitute, University of the Philippines, Diliman (e-mail: joyarmadamba@gmail.com)

F. J. Seriña is with Autodesk Inc., at Pittsburgh, Pennsylvania

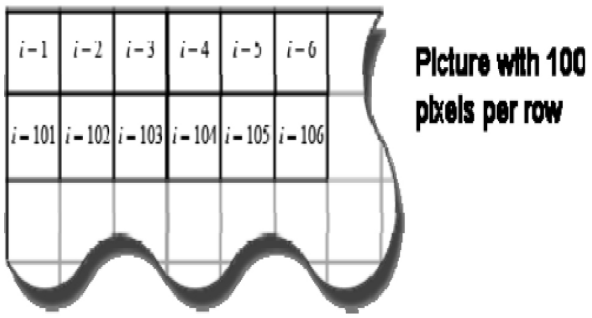


Fig 1. Scatter decomposition

Dynamic load balancing redistributes the tasks during processing. Techniques that use this type of load balancing can be further subdivided into two approaches: centralized and decentralized [6]. The centralized approach uses one machine as the controller (or master) of the system. It is responsible for task allocation and assignment. The decentralized approach uses all the machines in deciding where the next tasks will be given, with or without cooperation between machines. We focused on the centralized techniques as these would be more suitable for ray tracing. Some centralized techniques include the algorithm by Wang, et al. [7] that assigns tasks based on whether it is an I/O consuming task or a CPU compute task and the algorithm by Sidhu et al. [8] that uses particle swarm optimization. However, the first algorithm mentioned was not used as the assignment of tasks would introduce more variables in a heterogeneous system and the second is more suitable to a large-scale network. A third highly scalable centralized dynamic technique is Diffusion, which redistributes the tasks evenly when one of the slaves have completed earlier than the others [4]. Diffusion is started when a slave completes its task queue by sending a request signal to its neighboring machines. The neighbors reply with the number of tasks left in their queue. The average number of the remaining tasks is determined and the machines with a higher average give their excess tasks to the machines with a smaller queue. It has been shown that a hybrid technique (scatter decomposition with diffusion) resulted in greater efficiencies and speed up compared to them working alone [4]. However, scatter decomposition and diffusion only used homogenous computer architectures in their networks.

In order to normalize the performance of networks that use computers with heterogeneous architectures, an additional parameter is taken into consideration: speed index [5]. This determines the relative speed of one machine over another within the system. It is measured by determining the execution time of running a recursive function. Thus, distribution of tasks took advantage of which computer is faster by possibly assigning more tasks to these computers.

The goal of the study was to implement a new load balancing technique that uses a reservoir and compare its performance against existing load balancing algorithms. The study was implemented on a fully connected network with different computer hardware architectures. Figure 2 shows the hardware environment and the communications involved in the system. The hardware environment used heterogeneous commercial desktop computers with 1 designated as the master and the rest as slaves. It used the distributed memory architecture and communicated over a fully connected network.

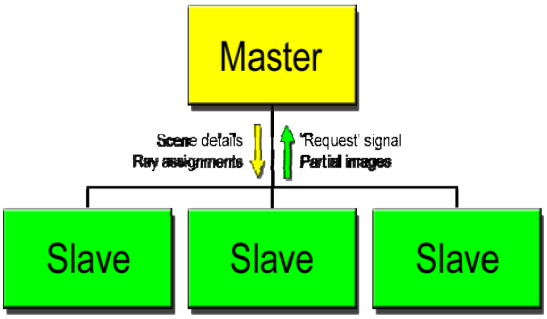


Fig 2. Implementation environment

The master was responsible for initializing the scene and distributing the tasks. It shall accept a 'request for more' signal from any slave and give it more tasks to do unless there's none left. The master accepts all the partial images from the slaves and compiles them as a single image. The slaves performed the ray tracing and requested for more tasks once its queue is empty.

II. METHODOLOGY

A. Reservoir Algorithm

The diffusion algorithm has a high communication overhead especially when the computers are in a fully connected network. When a slave requests for more rays, the master will ask each slave to send their task queue. The master averages the number of tasks and calls the slaves with task queue higher than the average to distribute their tasks to those who have less. Meaning, it redistributes tasks that were initially assigned already. This overhead is neglected since the amount of time it takes to send and receive data is much smaller than the rendering process. The reservoir algorithm will reduce the communication overhead by preventing any redistribution needed in the middle of the process. It does this by making the initial task distribution biased in such a way that faster computers receive more. Also, it does not distribute all the tasks during initial assignment. It keeps a certain number of tasks in a reservoir which will only be assigned when a slave requests for more.

There are three (3) major blocks in the Reservoir algorithm, as shown in Figure 3. The Reservoir Preparation uses the speed index of each slave to calculate the division of labor when biasing the task distribution. The steps for this procedure are shown next.

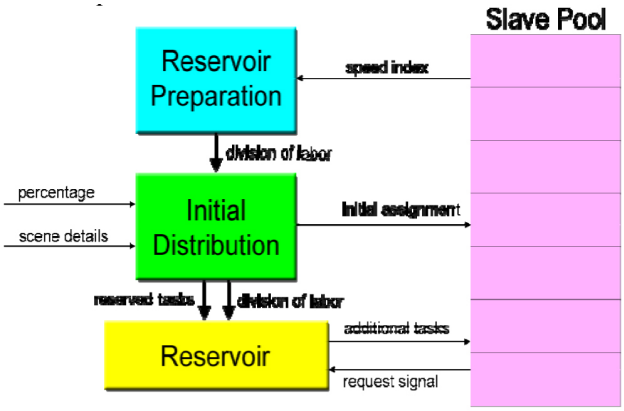


Fig. 3. Reservoir block diagram

Input: speed indices ($speed_1$, $speed_2$, ..., $speed_c$) and the number (c) of slaves
Output: sequence of percentage division of labor per slave (div_1, div_2, ..., div_c)

```
1.   procedure reservoir_preparation (speed, c)
2.       slowest : = speed_1
3.       for i : = 2 to c do
4.           if speed_i > slowest then
5.               slowest : = speed_i
6.       ratio_total : = 0
7.       for i : = 1 to c do
8.           begin
9.           ratio_i : = slowest / speed_i
10.          ratio_total : = ratio_total + ratio_i
11.          end
12.      for i : = 1 to c do
13.          div_i : = ratio_i / ratio_total
14.      return (div)
15.  end reservoir_preparation
```

The variable slowest is the speed index of the slowest computer within the network. This is searched for linearly in lines 2–5. In the sequence, $ratio_i$ is the theoretical equivalent amount of work that can be completed by $slave_i$ in the same time that the slowest slave completes 1 task—assuming that all tasks are equal. The sum of all ratio's are stored in ratio_total (lines 6–11) to be used as divisor in lines 12–13 to determine the division of labor.

The Initial Distribution block initially allocates specific amounts of the total tasks to the slaves. This initial number of tasks is determined by a quantity specified by the user called the initial task distribution percentage or initial percentage. Subsequently, a sequence of pixels is computed by multiplying the division of labor, initial percentage, and total number of tasks. These pixels are then given in a sequential manner. The first sequence of pixels is given to the first slave, the second sequence to the second slave, etc. The remaining unassigned tasks will be placed in the last block as reserved tasks. The steps for this procedure are shown next.

Input: initial task distribution percentage (p)
sequence (ray_1, ray_2, ..., ray_n) and number (n) of tasks
sequence ($slave_1$, $slave_2$, ..., $slave_c$) and number (c) of slaves
division of labor (div_1, div_2, ..., div_c) per slave
Output: undistributed tasks (*reserve*)

```
1.   procedure initial_distribution (p, ray, n, slave, c, div)
2.   limit : = p * n
3.   prev_limit : = 0
4.   for s : = 1 to c do
5.           begin
6.           next_limit : = prev_limit + (limit * div_s)
7.           for i : = prev_limit to next_limit do
8.                   begin
9.   AssignTask (ray_i, slave_s)
10.                  end
11.          prev_limit : = next_limit + 1;
12.          end
13.  reserve : = ray_limit+1, ray_limit+2, ... ray_n
14.          return (reserve)
end initial_distribution
```

When a slave finishes its assigned tasks, it will ask for more from the reservoir. In the final block of the algorithm, a fraction of the reserved tasks is given to the requesting slave based on the division of labor. The algorithm ends when there are no more reserved tasks, as shown below.

Input: requesting slave (*slave*)
division of labor of requesting slave (*div*)
sequence (ray_1, ray_2, ..., ray_n) and number (n) of undistributed tasks
Output: remaining undistributed tasks (*reserve*)

```
1.   procedure Reservoir (slave, div, ray, n)
2.           limit : = div * n
3.           for i : = 1 to limit do
4.                   AssignTask (rayi, slave)
5.           reserve : = raylimit+1, raylimit+2, ..., ray_n
6.           return (reserve)
end Reservoir
```

TABLE I
SAMPLE SIMULATION

Slave	Speed Index	Slowest	Ratio	Sum of Ratios	Percent division of labor	Ex 100 Tasks
1	1	Slave 3 with speed index 12	12	18.714	64.12% (65%)	65
2	3		4		21.37% (22%)	
3	12		1		9.16% (10%)	
4	7		1.714		5.34% (6%)	

All arithmetic operations are rounded up for the algorithm to converge. A test simulation is shown in Table 1. Slave 3 is determined to be the slowest with speed index 12. Ratio is the slowest speed index divided by the speed index of the slave under consideration. It represents how many tasks that computer can accomplish compared to the slowest assuming that all tasks are equal. For example, since slave 1 is 12 times faster compared to the slave 3, it could theoretically complete 12 tasks when slave 3 completes 1. The sum of these ratios will be used as a divisor to acquire the percent division of labor. The division of labor must be rounded up for the algorithm to converge.

On the last column, we assume slave 1 finished its tasks and there are 100 tasks in the reserve. Since the percent division of labor of slave 1 is 65%, 65 tasks will be given to it and the rest will remain as reserve until another slave completes its queue.

B. Phase I – Serial Ray Tracer (SRT)

In the first phase of the study, a functional serial ray tracer was built, which was used as the basis for the succeeding tracers. Ray tracing is a pixel by pixel type of rendering. Each pixel in the final image is calculated independently from those beside it. Looking at Figure 4, **E** refers to the viewer's eye which is looking at object **6**. Objects **6** and **4** are opaque spheres. Objects **3** and **9** are translucent planes and **LA** and **LB** are light sources. The different rays will be described later. A primary ray (from **E** to **3**) from the viewport travels in a straight line towards a pixel on the screen. This ray continues to the scene until it hits a surface. The algorithm that finds any intersection is called visible surface determination or intersection tests. Different algorithms were developed to find the most efficient visible surface determination algorithm. This is often a bottleneck in ray tracing algorithms as well as trying to parallelize it. Characteristics of the surface are taken into consideration to determine the reflection and refraction properties as well as its color at the point of intersection.

Shown in Figure 5 is the serial ray trace flowchart. The shading model used in the flowchart is a simplified hall shading model which consists of 4 terms. The first term is the low-level ambient light. The second term is a modified Phong Shading model which combines diffuse and specular term. The ambient term is neglected because the simplified hall shading model already considers it. The third term is the perfect specular reflection and the fourth is perfect specular transmission. The third and fourth terms are subject to recursion.

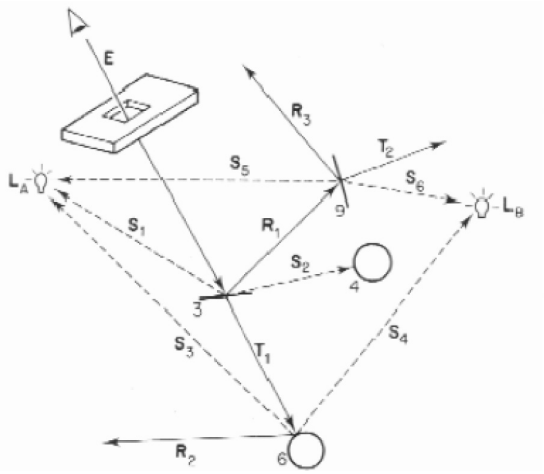


Fig. 4. Ray tracing paths

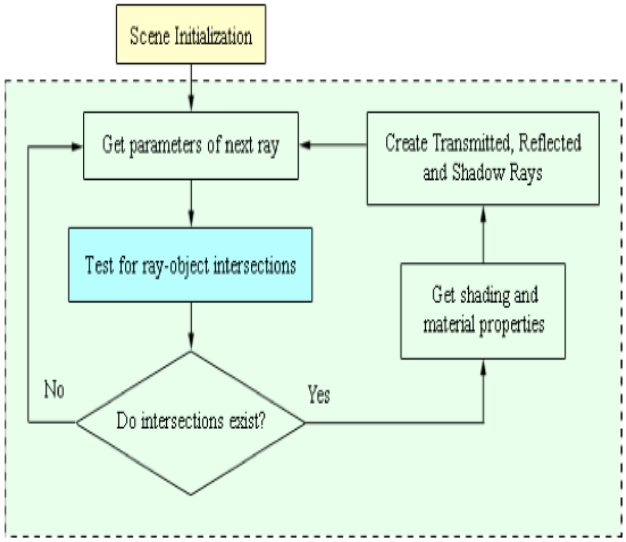


Fig. 5. Serial ray tracer

To make ray tracing faster, effort was given to the intersection test algorithm. The bulk of the running time of any ray tracer is devoted in finding the intersection between each ray and scene objects. To reduce the calculations, each object is assigned a bounding box—it is a box that tightly covers the space occupied by each object. The minimum and maximum x, y, and z coordinates of the object will be used as the dimensions of the bounding box. The 6 coordinates will correspond to the 6 faces of the box. When testing for the intersection between a ray and an object, the ray is tested against one face at a time.

The ray tracer could read the scene details from an external text file. The details include camera information, output image information, light sources, object (with surface) properties and environment variables. The output image information includes the width and height in pixels and the output file name, which has the extension of the output image type. The light sources are defined using its location and color. The object properties are defined differently. Surface properties have the same syntax for all types of primitives and are defined together with the object. The environment variables include the background color, ambient color and the maximum levels of reflection/transmission rays.

The serial ray tracer, using the minimum set of features, was able to render images with a single machine. It was able to render three (3) different types of primitive objects, namely spheres, boxes and one-sided planes, shown in Figure 6. Its resulting images have shown reflections, transmissions and shadows, similar to Figure 7.

Additional features such as super sampling and distributed ray tracing were also implemented to improve the quality of the pictures and increase the weight of the tasks. Super sampling is an anti-aliasing technique by shooting more rays per pixel. Distributed ray tracing slightly changes the angle of produced reflection, refraction and shadow rays to apply blurry phenomena such as gloss, blurry transparency, and penumbras. Shown in Figure 8 is a sample scene that shows all the basic features. A transparent sphere with properties similar to glass is included in the figure. Below it is a plane that reflects everything on top of it. There are multiple opaque left objects, 3 smaller blue spheres and a yellow box behind the transparent sphere. Two point-light sources are used, one behind the camera and another to the left of the box. Super sampling and distributed ray tracing were also implemented. The functionality of the SRT was then tested and verified.

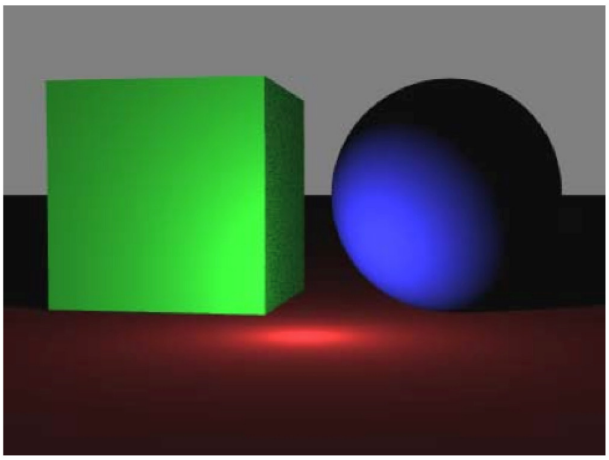


Fig. 6. Basic primitives

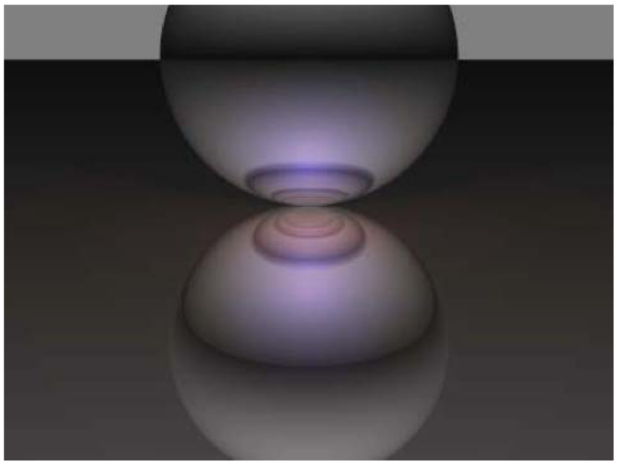


Fig. 7. Shiny sphere over a mirror

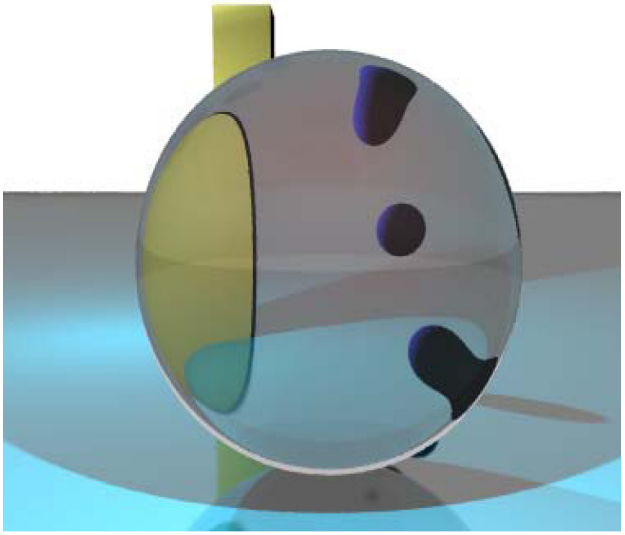


Fig. 8. Image created by SRT with basic features

C. Phase II – Parallel Ray Tracer

The next phase involved implementing the first of the three (3) parallel ray tracers (PRT). This PRT implemented only the scatter decomposition as a load balancing technique.

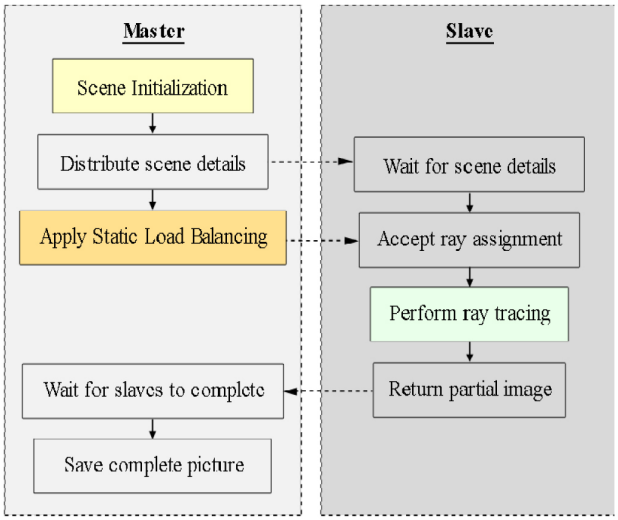


Fig. 9. Parallel ray tracer

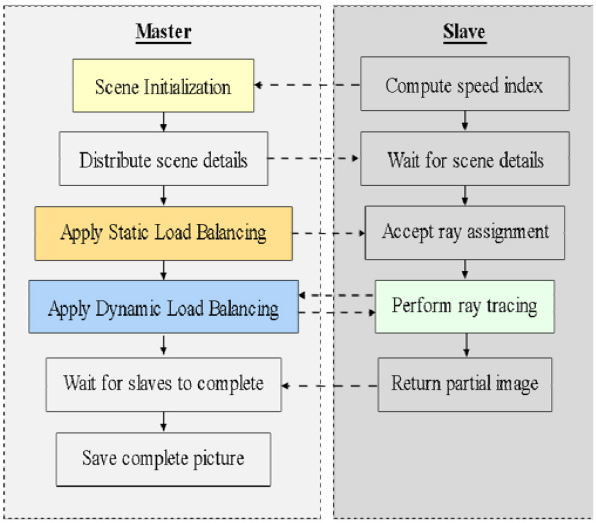


Fig. 10. Parallel ray tracer that uses dynamic load balancing

A master and slave side was created to distribute the tasks of the PRT. Additional modules were created for the master to pass the scene details to the slaves and for the slaves to compile the partial images for the master. The flowchart and points of communication are shown in Figure 9. Initialization is executed by the master while the slave executes ray tracing. The distribution of rays amongst the slaves, third block in the master side, is the static load balancing.

Different computers were used to test the parallelized ray tracer. The execution and idle time of each slave are noted in order to calculate efficiency and speed up. The results are tested against the results of phase III. The overall efficiency of the program is the sum of the execution times of each slave over the total elapsed time (execution and idle time) of all the slaves.

D. Phase III – Parallel Ray Tracer with Dynamic Load Balancing

The last two PRTs were implemented in this phase. The first PRT used the hybrid load balancing technique which combines scatter decomposition and diffusion. The second PRT implemented used the reservoir algorithm. Additional modules were also created, such as the module that computes the speed index and the module that commands slaves to receive/send tasks from/to other slaves. Figure 10 shows the block diagram of the PRT with dynamic load balancing.

E. Testing and Benchmarking

For the testing of the SRT, a series of test scenes were created that focused on the specific features of the ray tracer. This set of scenes was also rendered using an older version of a commercial ray tracer originally developed by Mental Images, the Mental Ray tracer. The resulting images were compared against those created by the SRT.

The test scenes created focused on the following features: shading, reflections, shadows, transmissions, and their combinations. Figure 11 shows one image that combines all of the features.

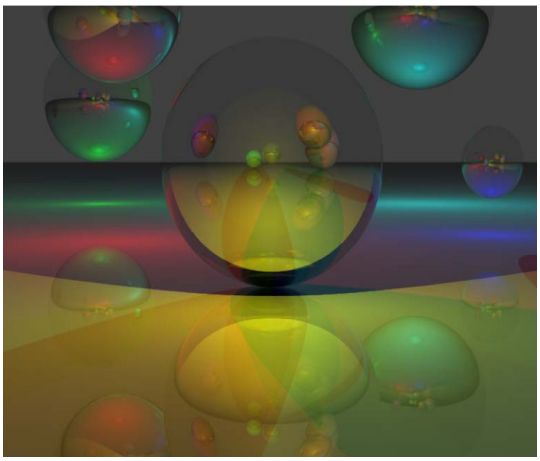


Fig. 11. Test scene that combined features

To compare the three (3) PRTs with the SRT, a complex scene (Figure 12) was rendered with varying parameters by all four ray tracers. This test scene approximated practical scenes since it has varying workloads throughout the image. The SRT was modified to calculate its speed index even if it was not used during its image processing. It was used later as a reference for comparison. For the PRTs, the total program execution times and slave speed index were recorded. The speed-up was computed by dividing the execution time of

the PRT over the execution time of the SRT that rendered the image with the same parameters. It was assumed that since the hardware environment used was exactly the same for all the tests, the speed indices of the machines within the network would be the same for all tracers.

The efficiency is computed as follows:

$$\eta = (\sum ET_S)/(ET_T \times n) \quad (1)$$

where ET_T is total elapsed time of the program, n is the number of slaves, and ΣET_S is a summation of the execution times of all slaves.

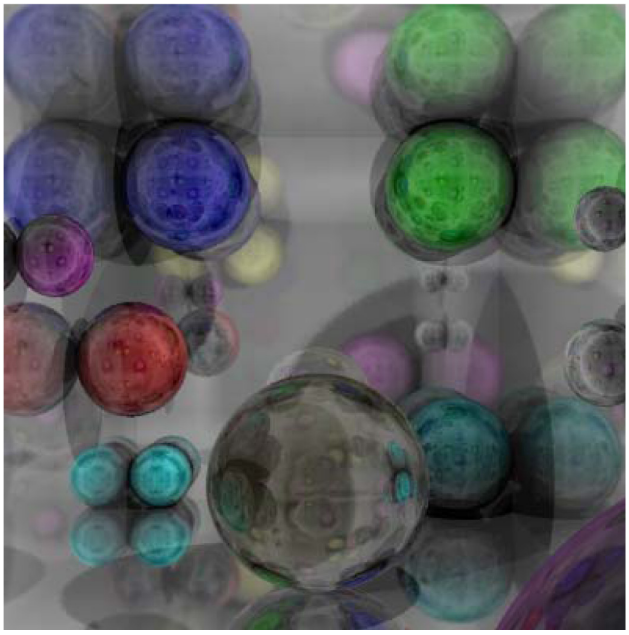


Fig. 12. Complex scene used as a benchmark.

Since the computers within the network do not have equal speeds, a new unit of measurement was used. This is the achieved speed-up (ASU) which relies on the validity of the speed index. The ASU is the ratio of the computed speed up compared to the maximum achievable speed up (MASU). Table 2 shows an example on how to compute the MASU and ASU. The theoretical speed-up is computed by dividing the speed index of the SRT over the speed index of the machine under consideration. The sum of all these theoretical speed ups would account for the MASU. If all the machines are homogenous, including the SRT, then the MASU is equal to the number of the slaves (ex. 4 homogenous slaves have 400% MASU). For the example below, the MASU is 341.79%. The ASU is then computed by dividing the speed up over the MASU.

TABLE 2
SAMPLE COMPUTATION OF ASU

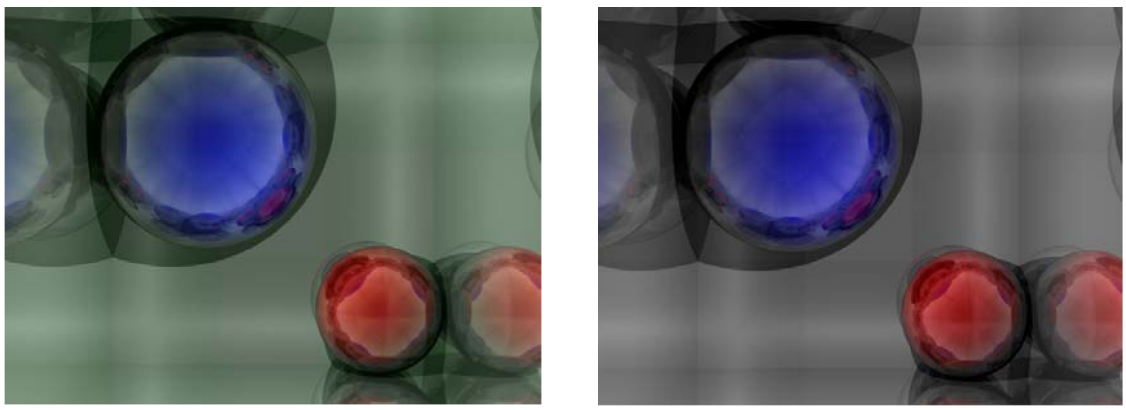
Machine	Speed Index (seconds)	Theoretical Speed Up (%)	Achievable Speed Up (%)
SRT	0.3	100	–
Slave 1	0.30525	98.28	28.75
Slave 2	0.644833	46.52	13.61
Slave 3	0.301333	99.56	29.13
Slave 4	0.307917	97.43	28.51

III. RESULTS AND ANALYSIS

A. Validity of Features

The SRT built from phase I was compared with another ray tracer using special scenes that were customized in order to exclude other features that the serial ray tracers could not perform. The following are a series of images of the different scenes rendered in both the SRT and Mental Ray Renderer.

As can be seen, Figure 13 has a significant difference—mainly, the hue of green visible in the whole scene. This is



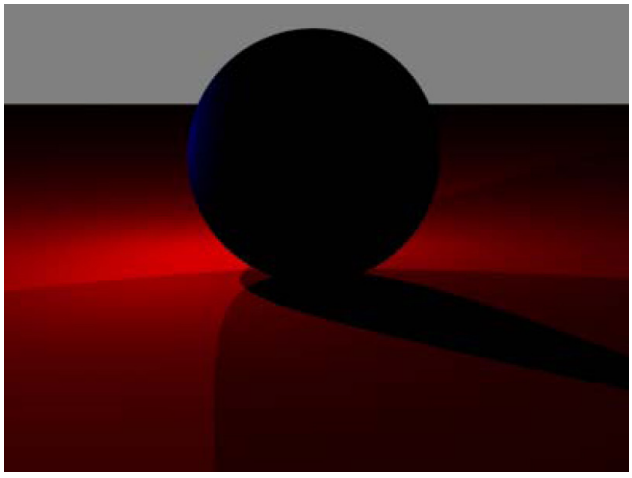
a. Mental Ray Render b. Serial Ray Tracer

Fig. 13. Shading equation and reflections

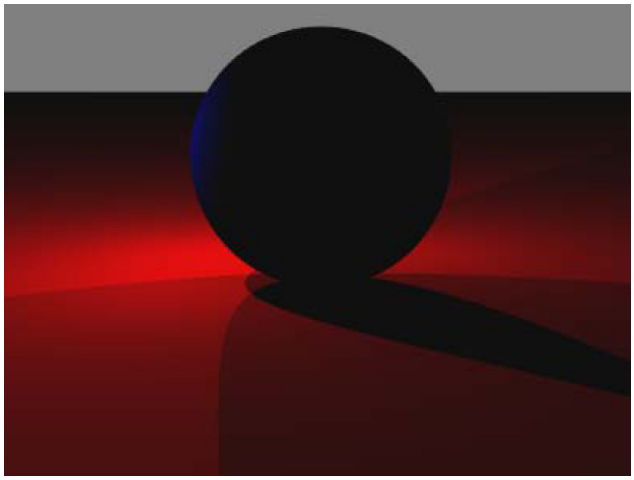
due to the set ambient light and the shading equation used. The Mental Ray Renderer applies a bigger weight to the ambient color compared to the serial ray tracer. Looking at the reflection and the color of the objects in the scene, it can be shown that other elements of the shading equation between the Mental Ray Renderer and the serial ray tracer is the same for the surface properties present.

In Figures 14 and 15, we can clearly see that the images produced are identical, showing the accuracy of the serial ray tracer in depicting shadows and transmissions.

Figure 16 shows that when the scene gets more complicated, the difference between the serial ray tracer and that of the Mental Ray Renderer becomes unnoticeable.

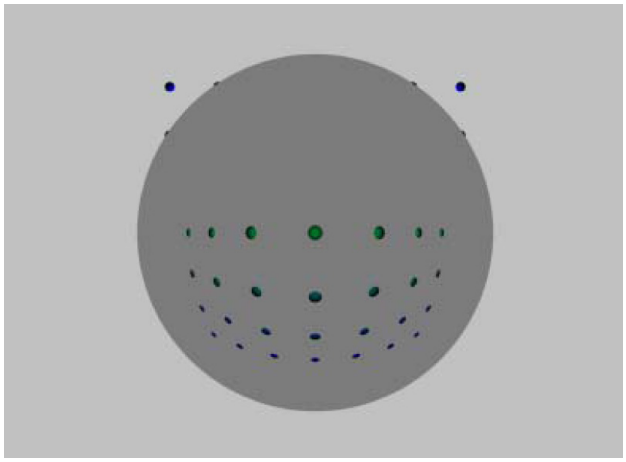


a. Mental Ray Render

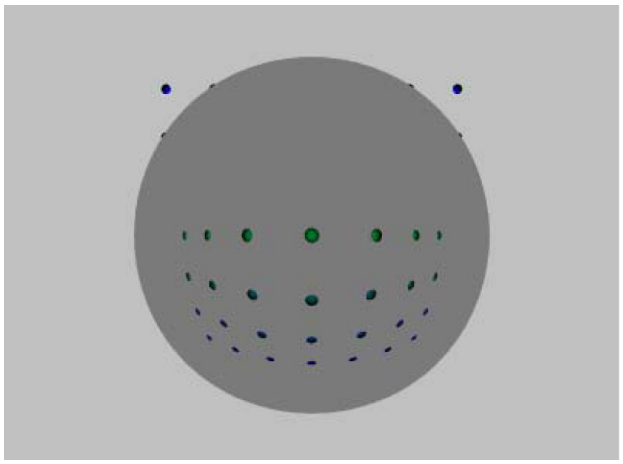


b. Serial Ray Tracer

Fig. 14. Shadows

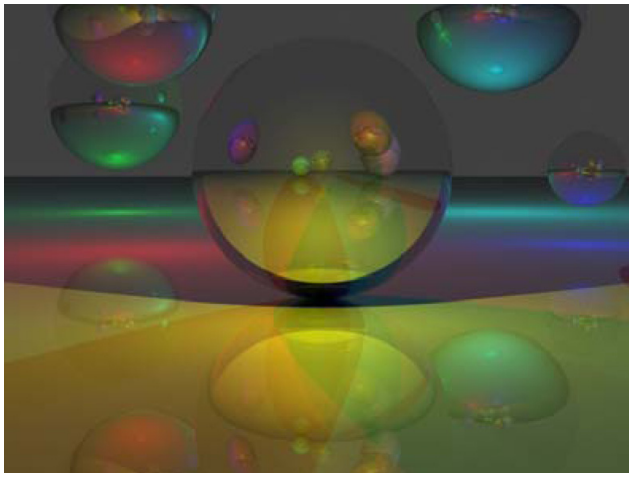


a. Mental Ray Render

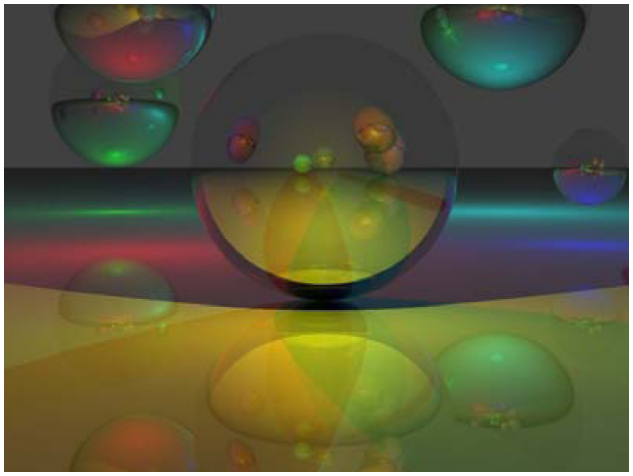


b. Serial Ray Tracer

Fig. 15. Transmissions



a. Mental Ray Render



b. Serial Ray Tracer

Fig. 16. Lights, reflections and shadows

B. Serial Ray Tracer Execution Times

One objective of this study was to build a PRT and compare the performance of the proposed algorithm with the existing load balancing techniques. Using the complex scene at Figure 12, Table 3 shows the execution time of the serial ray tracer, with an average speed index of 0.3, with different parameters. DRT refers to distributed ray tracing rate, SS is the super sampling rate, ET is the execution time in seconds. Since these four test cases were used all throughout the testing, they will be referenced as shown. The execution time is shown in HH:MM:SS. The four test cases do not have a constant increase in complexity of rendering with these parameters but as the execution time shows, test case C and D are much more difficult compared to A and B. This table will also serve as a reference for the speed up later on.

TABLE III

SERIAL RAY TRACER RESULTS

Test Case	DRT	SS	ET
A	1	8	4:14.51
B	1	16	8:34.16
C	2	8	5:22:5
D	2	16	10:46:12

C. Validity of Computing Speed Index

To measure the speed indices of the machines used in the network, a recursive function, the classic Towers of Hanoi, was executed. This function was chosen due to its size. It is small enough to minimize the overhead for its computation and reliable enough to imitate the execution of ray tracing while measuring the speed index. To verify this assumption, the two algorithms were executed and compared using the speed up, MASU, and ASU. Reservoir was used as the load balancing technique with an initial percentage set to 80%.

It can be seen in Figure 17 that the difference of values in the MASU and ASU is at 23.81% and 3.14% respectively. This means that these speed index-based values for the two functions differ by less than 10% of the higher value (42.38 for the MASU and 5.31 for the ASU). Therefore, the method used to approximate the speed index is relatively accurate.

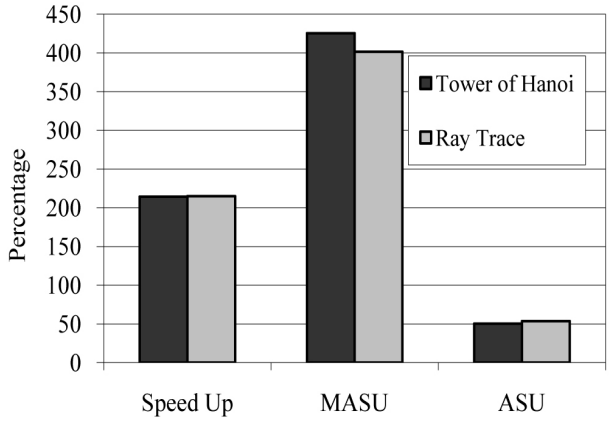


Fig. 17. Speed Indices of Algorithms

D. Maximum Speed Up for Reservoir

One difference of the reservoir algorithm over other load balancing techniques is the use of an additional parameter —initial task distribution percentage. In this test, the most effective initial percentage is determined for later use. The complex scene is rendered using test case C from Table 3 while varying the initial percentage. The speed up and efficiency of the PRT are shown in Figure 18.

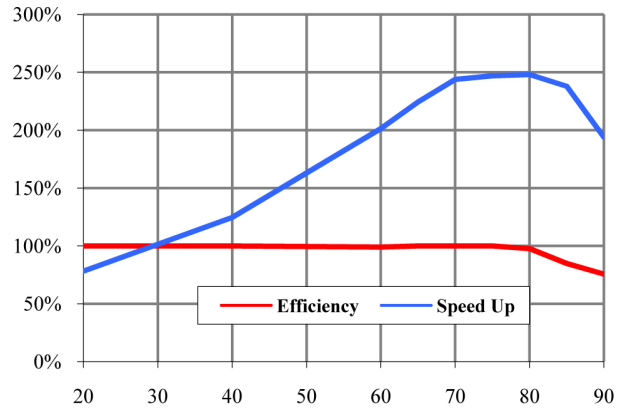


Fig. 18. Comparison of Efficiency and Speed Up of PRT Using Reservoir

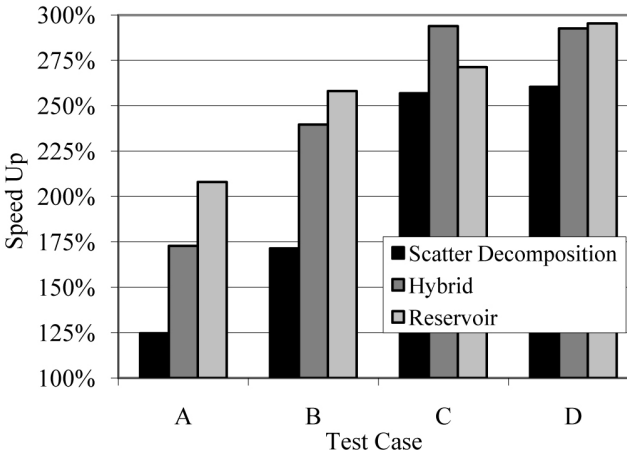


Fig. 19. Speed Up Comparison

Since the objective of parallelizing applications is to increase speed up, it can be shown that the best initial task distribution percentage for this environment is at 80%. Beyond 80%, the speed up decreases and it performs more similar to scatter decomposition both in efficiency and speed up. Below 80%, fewer tasks are initially distributed. These will make the slaves ask for more tasks earlier during the execution of the program. It is possible that the slaves tend to wait in line when requesting for additional tasks.

The efficiency, on the other hand, is consistently at 99% until the initial percentage is at 75%. Beyond that, it decreases. This is because the algorithm approaches the performance of the scatter decomposition implementation when using higher initial percentages. In these cases, a large number of tasks remain on the slaves and do not get redistributed even when the reservoir is empty and other slaves have completed their task queues.

The objective of parallelizing ray tracing is to shorten the execution time. It does not necessarily mean achieving the maximum efficiency. Thus, all PRT using reservoir used 80% as the initial task distribution percentage.

E. Comparison of Parallel Ray Tracers

Figure 19 shows the comparison of speed ups of the 3 PRT implementations. It can be seen in the results that the PRT with reservoir is better in most cases. The hybrid implementation is similarly high for the more difficult cases, approximately at 292% speed up. As expected, the PRT with scatter decomposition alone is the slowest of all implementations.

The speed up test is valid since the hardware environment is the same for all the different PRTs.

Figure 20 shows the comparison of efficiencies. It can be seen in the figure that the hybrid implementation is consistently performing at around 99.9%, better compared to all three PRTs. However, the difference with the reservoir implementation is small, at less than 10%. This difference can be attributed to the initial task distribution percentage, which was not chosen for its maximum efficiency.

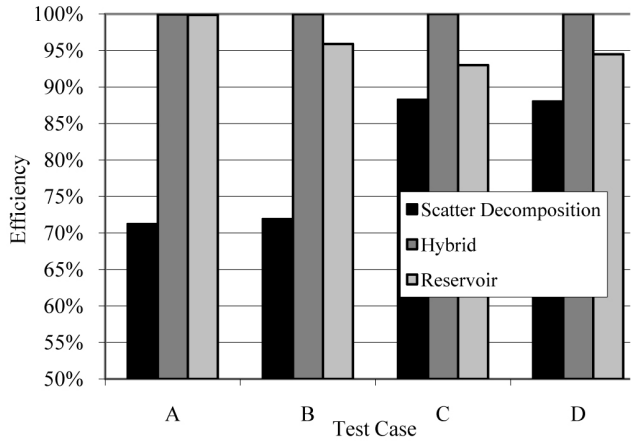


Fig. 20. Efficiency Comparison

Figure 21 shows the ASU of each PRT implementation. As seen in Figure 14, the reservoir implementation is better in most cases. This means that the algorithm was more able to maximize the hardware environment compared to the other implementations.

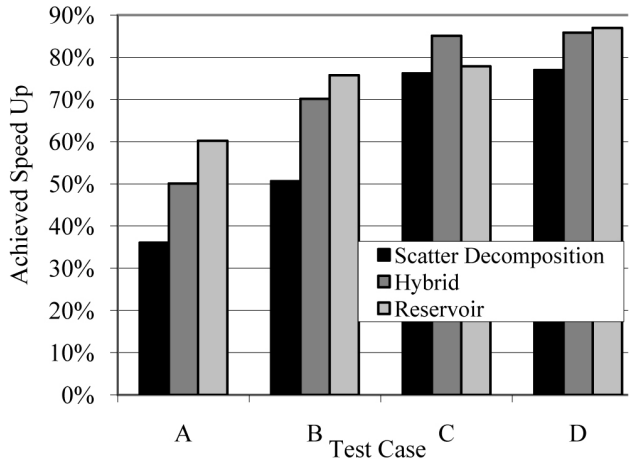


Fig. 21. Achieved Speed Up Comparison

IV. Conclusion and Recommendations

The results have shown that when using a reservoir, high speed-up is achieved at 80% initial task distribution while high efficiency is achieved between 20–75% initial task distribution. With speed-up as the priority, reservoir achieves at most 87% better speed-up than the scatter decomposition and 36% than the hybrid technique. It also has at most 28% better efficiency than the scatter decomposition. The reservoir algorithm achieved at most 25% better achieved speed-up (ASU) than the scatter decomposition and 10% than the hybrid. Thus, in conclusion, under the scenarios tested and the methodologies used, the load balancing technique that uses a reservoir is a good alternative technique for load balancing in parallel ray tracing. This technique has a simpler implementation than diffusion since assignment of tasks is only done once and no tasks are moved between slaves.

Even if complex scenes were used in testing, we recommend that real practical scenes be used as benchmarks. Personal evaluation of these scenes can be used as an additional test to determine the realism of the images. The concept of applying a bias to the distribution using the speed index can also be applied for other load balancing techniques that use heterogeneous computers. With this type of implementation, a possible increase to the speed up, ASU and the efficiency may be seen. Also, the current hardware environment is limited to a single master and four slaves. It is possible that the algorithm may perform differently with varying number of slaves. In this case, the ASU can be used as the primary measurement of comparison as the heterogeneity of the system will be more prominent.

References

[1] Glassner, A. *Introduction to Ray Tracing*. Morgan Kaufmann Publishers, Inc., 1989.

[2] J. Hurley. "Ray Tracing Goes Mainstream," *Intel Technology Journal*. Volume 9, Issue 2, p. 107, May 19, 2005.

[3] Watts, J., Taylor, S. "A Practical Approach to Dynamic Load Balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol 9, no. 3, March 1998.

[4] Arvo, J. & Heirich, A. "A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing." Kluwer Academic Publishers, Boston. California Institute of Technology, 1998.

[5] Fellner, D., Schafer, S., and Zens, M. "Photorealistic Rendering in Heterogeneous Networks," *Advances in Parallel Computing*, vol. 12, pp. 113–120, 1998.

[6] Sahoo, B. "Dynamic Load Balancing Strategies in Heterogeneous System (Doctoral dissertation)," Retrieved from *ethesis.nitrkl.ac.in/5642/1/dlbmain.pdf*, 2013.

[7] Wang, H. et al. "An Innovate Dynamic Load Balancing Algorithm Based on Task Classification," *IJACT: International Journal of Advancements in Computing Technology*, vol. 4, no. 6, pp. 244–254, 2012.

[8] Sidhu, M., Thulasiraman, P., and Thulasiram, R. "A Load-rebalance PSO Heuristic for Task Matching in Heterogeneous Computing Systems," *IEEE Symposium on Swarm Intelligence (SIS)*, pp. 180–187, 2013.