Robert Hughes
Parallel Computing
4/30/18
Ray-Tracing Parallel Computing Project

Problem:

The basis of this project was to take a simple ray tracing program parallelize it and then analyze the speedup and efficiency. For this problem it is important to understand first what ray tracing is. Ray tracing is a graphic rendering technique. The way that an image will be made is by tracing the path of a simulated light as the pixels in an image plane. When the light hits the simulated objects, given that objects characteristics the light will behave in different ways to make the objects appear. The reason for using this method is that it creates a higher level of realism in the rendering images. However compared to other rendering techniques ray tracing takes more computational power. Because of this high computation cost ray tracing is not used in video games and real time rendering. However because of the high level of realism ray tracing is used in movies. One of the most well known examples is Cars. For a comparison the rasterization rendering will be compared. Rasterization is to create a bunch of virtual triangles to build the 3-D models. The computer takes these triangles and converts them into pixels or dots on a 2-D screen. It then uses shading and other approximations to “guess” as how the object should look. This technique is often used in video games because it is very fast and has a decent amount of realism. Ray tracing however provides this next level of realism because the objects are illuminated like they are in real life. Light can be blocked, refracted, reflected and have other things happen to it so that the objects act like they would in the real world. Figure 1 to the right shows an example of a ray traced image. This image was rendered using ray tracing and has an incredible level of realism.

Figure 1: Ray Traced Image

Early examples of ray tracing can be dated back to 1969 when an IBM technician described a technique for shading machine renderings of solids. It was not until 1977 that the technique was formed and documented. Then in 1984 the some employees from LucasFilm concluded that ray tracing could improve film-making. From there ray racing has improved and gotten better and better to the point where it is used in in lots of movies. With increasing capabilities of GPUs and more advanced computers ray tracing is faster and faster to complete. Currently NVIDIA, Microsoft, AMD, and other companies are working on real-time ray tracing to be used in video games and to speedup the overall process.

Figure 2: Threadwork Function with indexed array

Design:

I first stumbled upon ray tracing because I found an interesting article on the subject and NVIDIA’s current research in the field. Upon doing some more research there was a guy that decided to create ray tracer that would fit on the back of a business card. This was interesting because the code is usually fairly long and complicated and from there I decided that I would do ray racing as my parallel project. I decided to work on this project because it is such a good

candidate for parallelization. It is a good candidate because it is high computation cost. The way that I decided to implement the parallelization for this program was trough using pthreads. This program is a good candidate for pthreads because all the pixels are stored in a single array. By splitting up the work and having the threads store their computations in the array will be the best way. This method requires no critical sections because all the threads are accessing different parts of the array. I can then create a threadwork function and inside just use a for loop that has my_start and my_end for each thread to divide up the work. Figure 2 shows the for loop that the threads will run based on their indexes.

Experiments:

For my experiment I decided to run a couple of different experiments. First I examined what happened with different number of balls. I then examined what happened with different number of lights of on the set number of 5 balls. Figure 3 shows the output of the program and the various ball counts that were found. The program wrote out the ball into a .ppm file so I used a Gimp, photo editing software to open the photo and see if it worked. To run the different balls it was not hard. First I had to update the array that held all the balls to include another element, then I had to allow the loop to run once more, finally I had to initialize one more ball and give it various different qualities like what material it was made of, its positions, radius and the color. The material determined how reflective and the shading that the ball recieved. In the next part of the experiment with the different amount of lights. Seen in figure 4 is the different lights that were added on the 5 balls you can see that the shading and the reflections are different

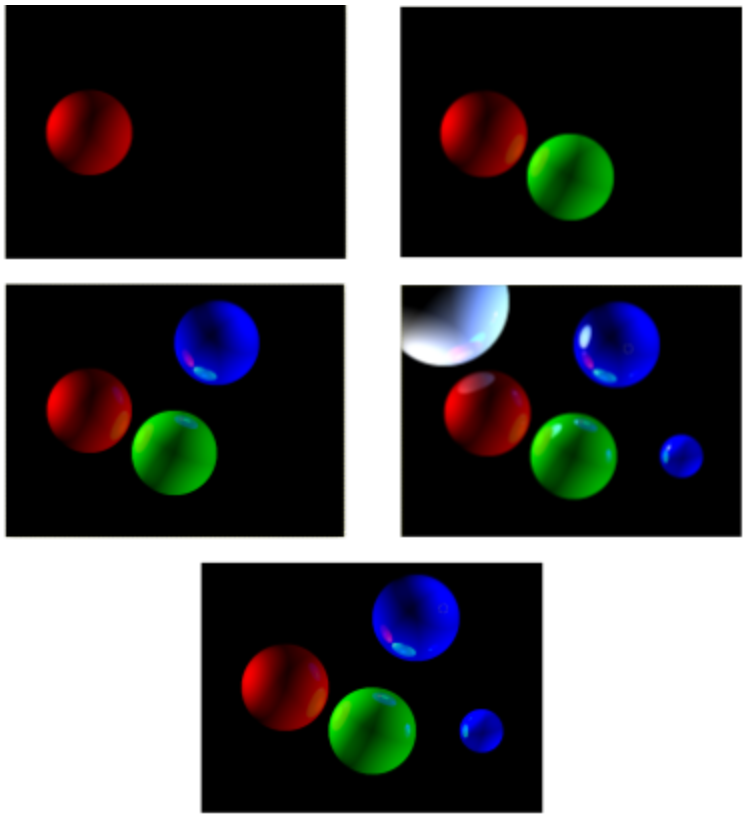

Figure 3 Images of Various ball counts outputted by my program

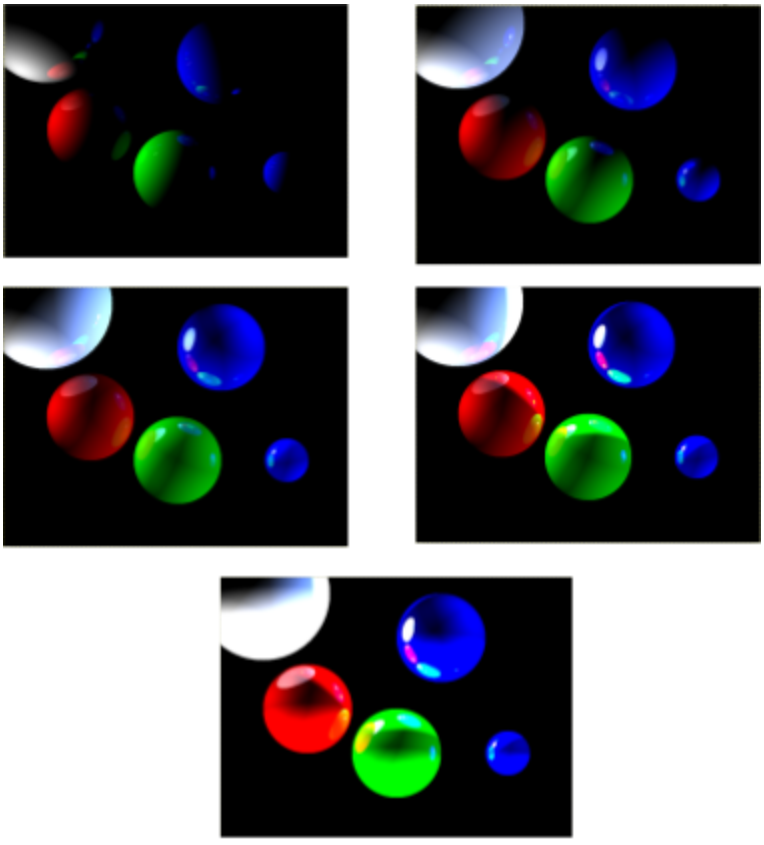

Figure 4 Different Lighting From Right to Left number of lights goes up by one

given the different number of lights. To change the amount of light I had to update the light array that held the lights and their properties. Then I added one more light element to the light for loop. Finally I initialized the new light by setting the xyz coordinates, and the amount of light that there would be. Changing these things was relatively easy. As an example The code can show us what goes into the light and the sphere structure. This program used a number of different structures one for the light and one for the ball. It also included one for the light ray as a way to track its properties. There was also a structure for the material, the material was used to determine the properties such as the transparency, the shininess and the reflective properties of the ball. After setting all these different parts up I ran the program in a number of different configurations. I ran it from 1-128 different threads and these threads on the various setups discussed above with the balls and the lights.I then took timings of all these different configurations and tabulated them.Table 1 and Table 2 shows the results that were recorded they can be found on the next page. The reason for the different number of lights is that looking at how the different number of light rays and balls is to see if the number of light interactions on a fixed number of balls changes the time compared to the number of balls. All these were timed on a 4 core machine meaning we will see a max speed and efficiency around 8 threads.

Robert Hughes
Parallel Computing
4/30/18
Ray-Tracing Parallel Computing Project

Analysis:

From the timing data I found both the efficiency and the Speedup. This is what I used to compare the different experiments and see how they compare to one another and what will be more optimal for the program to run. Looking a the figures below we can see this Data. For each of the 2 experiments there is the runtime graph, the speedup, and the efficiency. On each of these graphe we can find the all of the sub experiments on the graph labeled. It is interesting

to see that the time to run with different numbers of lights does not change the time significantly but this makes sense because the light interaction with the ball is what we are interested in and the lights do not need to be formed directly and all they require is some math operations to be performed which the computer can perform very quickly. We do see that there is a limit to how

fast the computations happen. The max does match with what we thought, it occured around 8 threads Past this threshold the speedup ad the efficiency starts to go down. The balls have a more obvious change in the time based on how many there are. All the balls follow the same curve similar to the lights just spread more apart based on thor computation changes. We also see that the speedup max occurs at 8-16 threads which is consistent based on the architecture of the machine that was being used. We also see a steep decline in efficiency when the limitation of the machine is surpassed. From both of these 2 experiments I conclude that this is a weakly scalable program. I conclude this because there is an apparent speedup however the efficiency starts to go down. It is worth noting that the efficiency does not decrease too much until the cores are overworked on the thread count where it tanks down to near 0. There are a couple different ways that I think this code could be modified so that the efficiency does not go down and the program becomes strongly scalable. I think that the best way would be to use a GPU to run this code rather than CPU the GPU could be much more efficient and could assign threads to each pixel in the image, this would increase speedup and efficiency. Another way that this could be more efficient is that if the program was changed so that instead of each thread taking a part of the picture it could instead take a light source or an object and just monitor how that structure should behave. This program would also work on an MPI system but

would have to be a more complex image for the speedup to override the cost of communication. As GPUs get better and more efficient algorithms are created ray tracing will be used more and more often. It would not be surprising to see real time ray tracing within the next 10 years.

Implementation:

This program is not too hard to build I grabbed the serial code from: https://github.com/PurpleAlien/Raytracer . There are lots of different projects involving ray tracing that this github has. He even has one that includes pthreads similar to mine except more complex. It allows users to pass in .3ds files so that they can add different figures to their rendered image. I just took the basic c ray tracer and changed it so that it would be in parallel. The reason that the image used rendered spheres is because they are the easiest to implement. Other shapes require more complex calculations and structures that I unfortunately did not have time to implement for this project. There are many ray tracers that are available online and many have different features. Luckily for me the program implementation came along well and anyone who completed the Parallel Computing course would be able to implement this program in parallel. Code for this project can be found at the end of the program and will also be uploaded to canvas as a .c file. The tables and the graphs lowered the in quality when I put then into this file so I will upload those as well.

Robert Hughes
Parallel Computing
4/30/18
Ray-Tracing Parallel Computing Project

```
/* A simple ray tracer */


#include <stdbool.h> /* Needed for boolean datatype */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <semaphore.h>
#include "timer.h"
#include <unistd.h>     // usleep()

// global constant for the max number of threads
const int MAX_THREADS = 1024;

#define min(a,b) (((a) < (b)) ? (a) : (b))

/* Width and height of out image */
#define WIDTH  800
#define HEIGHT 600




/* structure passed as argument to new threads */
typedef struct
{
   long rank;
} THREAD_ARG;

/* The vector structure */
typedef struct{
     float x,y,z;
}vector;
```

```
/* The sphere */
typedef struct{
        vector pos;
        float  radius;
    int material;
}sphere;

/* The ray */
typedef struct{
        vector start;
        vector dir;
}ray;

/* Colour */
typedef struct{
    float red, green, blue;
}colour;

/* Material Definition */
typedef struct{
    colour diffuse;
    float reflection;
}material;

/* Lightsource definition */
typedef struct{
    vector pos;
    colour intensity;
}light;

/* Subtract two vectors and return the resulting vector */
vector vectorSub(vector *v1, vector *v2){
    vector result = {v1->x - v2->x, v1->y - v2->y, v1->z - v2->z };
    return result;
}
```

```
	/* Multiply two vectors and return the resulting scalar (dot
product) */
	float vectorDot(vector *v1, vector *v2){
	    return v1->x * v2->x + v1->y * v2->y + v1->z * v2->z;
	}

	/* Calculate Vector x Scalar and return resulting Vector*/
	vector vectorScale(float c, vector *v){
		vector result = {v->x * c, v->y * c, v->z * c };
		return result;
	}

	/* Add two vectors and return the resulting vector */
	vector vectorAdd(vector *v1, vector *v2){
		vector result = {v1->x + v2->x, v1->y + v2->y, v1->z + v2->z
};
		return result;
	}



	/*Global Variables that all of the threads wil need access to*/
	   int num_threads = 0; // number of threads to be set by the
command line
	   double time_3 = 0;

	    material materials[4];


	    sphere spheres[5];


	    light lights[3];


	    /* Will contain the raw image */
	    unsigned char img[3*WIDTH*HEIGHT];
```

```
	/* Check if the ray and sphere intersect */
	bool intersectRaySphere(ray *r, sphere *s, float *t){

		bool retval = false;

		/* A = d.d, the vector dot product of the direction */
		float A = vectorDot(&r->dir, &r->dir);

		/* We need a vector representing the distance between the start
of
		  * the ray and the position of the circle.
		  * This is the term (p0 - c)
		  */
		vector dist = vectorSub(&r->start, &s->pos);

		/* 2d.(p0 - c) */
		float B = 2 * vectorDot(&r->dir, &dist);

		/* (p0 - c).(p0 - c) - r^2 */
		float C = vectorDot(&dist, &dist) - (s->radius * s->radius);

		/* Solving the discriminant */
		float discr = B * B - 4 * A * C;

		/* If the discriminant is negative, there are no real roots.
		  * Return false in that case as the ray misses the sphere.
		  * Return true in all other cases (can be one or two
intersections)
		  * t represents the distance between the start of the ray and
		  * the point on the sphere where it intersects.
		  */
		if(discr < 0)
			retval = false;
		else{
			float sqrtdiscr = sqrtf(discr);
```

```
            float t0 = (-B + sqrtdiscr)/(2);
            float t1 = (-B - sqrtdiscr)/(2);

            /* We want the closest one */
            if(t0 > t1)
                t0 = t1;

            /* Verify t1 larger than 0 and less than the original t */
            if((t0 > 0.001f) && (t0 < *t)){
                *t = t0;
                retval = true;
            }else
                retval = false;
        }

    return retval;
    }

    /* Output data as PPM file */
    void saveppm(char *filename, unsigned char *img, int width, int
height){
        /* FILE pointer */
        FILE *f;

        /* Open file for writing */
        f = fopen(filename, "wb");

        /* PPM header info, including the size of the image */
        fprintf(f, "P6 %d %d %d\n", width, height, 255);

        /* Write the image data to the file - remember 3 byte per pixel
*/
        fwrite(img, 3, width*height, f);

        /* Make sure you close the file */
        fclose(f);
    }
```

```
void processCommandLine(int argc, char* argv[]) {
     num_threads = strtol(argv[1], NULL, 10);
     if (num_threads <= 0)
        num_threads = 1;
    if (num_threads >= MAX_THREADS)
        num_threads = MAX_THREADS;
    printf("num [%s] threads: %ld\n",argv[1], num_threads);

}


void *threadWork(void* argstruct)
{
    ray r;
    /* unpack the thread arguments */
    long my_rank = ((THREAD_ARG*)argstruct)->rank;
     int num_height = HEIGHT/num_threads;
     int my_start = my_rank * num_height;
     int my_end = my_start + num_height;
     if(HEIGHT % num_threads != 0 && my_rank == num_threads -
1){//statement to fix the rows if the height is not divisible by 2^n
      int temp = HEIGHT - my_end;
      my_end = my_end + temp; // add the last portion to last thread
so all pieces are covered

      }

      printf("I have a rank! %d\n", my_rank);


      int x, y;
      for(y=my_start;y<my_end;y++){
          for(x=0;x<WIDTH;x++){

              float red = 0;
              float green = 0;
```

```
                    float blue = 0;

                    int level = 0;
                    float coef = 1.0;

                    r.start.x = x;
                    r.start.y = y;
                    r.start.z = -2000;

                    r.dir.x = 0;
                    r.dir.y = 0;
                    r.dir.z = 1;

                    do{
                        /* Find closest intersection */
                        float t = 20000.0f;
                        int currentSphere = -1;

                        unsigned int i;
                        for(i = 0; i < 5; i++){
                            if(intersectRaySphere(&r, &spheres[i], &t))
                                currentSphere = i;
                        }
                        if(currentSphere == -1) break;

                        vector scaled = vectorScale(t, &r.dir);
                        vector newStart = vectorAdd(&r.start, &scaled);

                        /* Find the normal for this new vector at the point
of intersection */
                        vector n = vectorSub(&newStart,
&spheres[currentSphere].pos);
                        float temp = vectorDot(&n, &n);
                        if(temp == 0) break;

                        temp = 1.0f / sqrtf(temp);
                        n = vectorScale(temp, &n);
```

```
                    /* Find the material to determine the colour */
                    material currentMat =
materials[spheres[currentSphere].material];

                    /* Find the value of the light at this point */
                    unsigned int j;
                    for(j=0; j < 3; j++){
                        light currentLight = lights[j];
                        vector dist = vectorSub(&currentLight.pos,
&newStart);
                        if(vectorDot(&n, &dist) <= 0.0f) continue;
                        float t = sqrtf(vectorDot(&dist,&dist));
                        if(t <= 0.0f) continue;

                        ray lightRay;
                        lightRay.start = newStart;
                        lightRay.dir = vectorScale((1/t), &dist);

                        /* Lambert diffusion */
                        float lambert = vectorDot(&lightRay.dir, &n) *
coef;
                        red += lambert * currentLight.intensity.red *
currentMat.diffuse.red;
                        green += lambert * currentLight.intensity.green
* currentMat.diffuse.green;
                        blue += lambert * currentLight.intensity.blue *
currentMat.diffuse.blue;
                    }
                    /* Iterate over the reflection */
                    coef *= currentMat.reflection;

                    /* The reflected ray start and direction */
                    r.start = newStart;
                    float reflect = 2.0f * vectorDot(&r.dir, &n);
                    vector tmp = vectorScale(reflect, &n);
                    r.dir = vectorSub(&r.dir, &tmp);
```

```
                    level++;

                }while((coef > 0.0f) && (level < 15));

                img[(x + y*WIDTH)*3 + 0] = (unsigned
char)min(red*255.0f, 255.0f);
                img[(x + y*WIDTH)*3 + 1] = (unsigned
char)min(green*255.0f, 255.0f);
                img[(x + y*WIDTH)*3 + 2] = (unsigned
char)min(blue*255.0f, 255.0f);
            }
        }


      time_3 = getProcessTime();


    return NULL;
    }









    int main(int argc, char *argv[]){
       //materials definition
       materials[0].diffuse.red = 1;
        materials[0].diffuse.green = 0;
        materials[0].diffuse.blue = 0;
        materials[0].reflection = 0.2;
```

```
            materials[1].diffuse.red = 0;
            materials[1].diffuse.green = 1;
            materials[1].diffuse.blue = 0;
            materials[1].reflection = 0.5;

            materials[2].diffuse.red = 0;
            materials[2].diffuse.green = 0;
            materials[2].diffuse.blue = 1;
            materials[2].reflection = 0.9;

           materials[3].diffuse.red = 1;
            materials[3].diffuse.green = 1;
            materials[3].diffuse.blue = 1;
            materials[3].reflection = 0.5;


            //sphere definition
            spheres[0].pos.x = 200;
            spheres[0].pos.y = 300;
            spheres[0].pos.z = 0;
            spheres[0].radius = 100;
            spheres[0].material = 0;

            spheres[1].pos.x = 400;
            spheres[1].pos.y = 400;
            spheres[1].pos.z = 0;
            spheres[1].radius = 100;
            spheres[1].material = 1;

            spheres[2].pos.x = 500;
            spheres[2].pos.y = 140;
            spheres[2].pos.z = 0;
            spheres[2].radius = 100;
            spheres[2].material = 2;

            //added sphere
            spheres[3].pos.x = 650;
```

```
        spheres[3].pos.y = 400;
        spheres[3].pos.z = 0;
        spheres[3].radius = 50;
        spheres[3].material = 2;

       spheres[4].pos.x = 100;
        spheres[4].pos.y = 40;
        spheres[4].pos.z = 0;
        spheres[4].radius = 150;
        spheres[4].material = 3;



        //light definitions

        lights[0].pos.x = 0;
        lights[0].pos.y = 240;
        lights[0].pos.z = -100;
        lights[0].intensity.red = 1;
        lights[0].intensity.green = 1;
        lights[0].intensity.blue = 1;

        lights[1].pos.x = 3200;
        lights[1].pos.y = 3000;
        lights[1].pos.z = -1000;
        lights[1].intensity.red = 0.6;
        lights[1].intensity.green = 0.7;
        lights[1].intensity.blue = 1;

        lights[2].pos.x = 600;
        lights[2].pos.y = 0;
        lights[2].pos.z = -100;
        lights[2].intensity.red = 0.3;
        lights[2].intensity.green = 0.5;
        lights[2].intensity.blue = 1;
```

```
        long t_rank;
        pthread_t* thread_handles;
        THREAD_ARG* thread_arguments;
        /* get initial clock time */
        double time_1 = getProcessTime();
         //get the number of threads fom the argument line
         processCommandLine(argc, argv);

         /* allocate thread handles */
        thread_handles =
            (pthread_t*)malloc(num_threads*sizeof(pthread_t));
        /* allocate thread argument structures */
        thread_arguments =
            (THREAD_ARG*)malloc(num_threads*sizeof(THREAD_ARG));

         double time_2 = getProcessTime();




         /* START OF CODE TO BE PARALLELIZED */
      for (t_rank = 0; t_rank < num_threads; t_rank++)
        {
            thread_arguments[t_rank].rank = t_rank;
            pthread_create(&thread_handles[t_rank],
                           NULL,
                           threadWork,
                           (void*) &(thread_arguments[t_rank])
                           );
        }
    printf("Main thread: All threads have been created.\n");



          /* wait for all threads to finish */
        for (t_rank = 0; t_rank < num_threads; t_rank++)
            pthread_join(thread_handles[t_rank], NULL);
```

```
        /* END OF CODE TO BE PARALLELIZED */


        saveppm("imageParallel.ppm", img, WIDTH, HEIGHT);

       printf("data input time: %f seconds\n", time_2-time_1);
        printf("computation time: %f seconds\n", time_3 - time_2);
        printf("total process time: %f seconds\n", time_3 - time_1);

    return 0;
    }
```