15418 Final Project Report

Parallel Monte Carlo Path Tracing

Cassidy Diamond (cdiamond) and Akintayo Salu (asalu)

URL: https://dialmond.github.io/15418-project/
GitHub code repo: https://github.com/dialmond/15418-project

Summary

We adapted and expanded upon the code for a sequential ray tracer, then parallelized it using OpenMP, MPI, and CUDA. We ran a multitude of experiments to analyze the performance of our implementations, including on the GHC Cluster machines and on the Pittsburgh Super Computer. Our MPI implementation achieved up to a 61.94x speedup on 64 cores, while our OpenMP implementation achieved up to 47.98x speedup on 64 cores (for reasons we will analyze in this report). With these results, we were able to demonstrate a live, real-time dynamic rendering of a raytraced scene at our poster presentation.

Background

Ray tracing is a method in computer vision to develop realistic scenes by mimicking the physical properties of light. Rays are shot from the camera into a scene, colliding with objects and bouncing around. Each collision reveals new information about the color of a pixel of the image's viewport, and collisions with various materials have different effects on the behavior of the ray (for example, scattering it further according to simulated properties of the material, or totally absorbing it). We trace the path of the ray and its collisions to get the final color of a pixel.

For each pixel, we shoot out numerous rays (according to the number of samples per pixel), randomly offset around the region we're projecting onto, then average the results together to get the final color. This a Monte Carlo algorithm designed to efficiently simulate the infinite integral of the path of all light on a scene. Each sampled ray lands somewhere in the scene, concentrated around where the pixel projects onto but with a little bit of randomness. For each ray, we find if it lands upon an object in our world – if it doesn't, we return the background color of the scene, which is either lit or totally dark – and if it does, we get the color of the material that we hit or the light the material emits. Most materials reflect light, and in different ways. In our program, we implemented lambertian materials which scatter light uniformly; metals which reflect light according to the angle of incidence, potentially imperfectly; and dielectric materials (like glass or water) that refract rays. For these materials, we get what portion of light they reflect, then recursively follow the path of the reflected ray into the world, where they may hit other objects and repeat the process up to some fixed depth. Finally, we average the color of all of our rays to get the final color for a pixel, and leading to the somewhat emergent properties of things like shadows, reflections, aliasing, bending light – elements of our vision that we experience in our real world.

The problem with ray tracing is that an enormous amount of computation must be done for each scene. All of the millions of pixels in an image have – in our implementation – up to hundreds of samples each. This naturally justifies the use of parallelism. However, due to the recursive and dynamic nature of ray tracing, where even rays offset by the same pixel can have drastically different paths due to the randomness of things like scattering, it becomes difficult to appropriately balance this work between processors. Furthermore, we experienced difficulties with memory sharing of the world information between threads, which tells rays which objects they hit, which materials those objects have, and what to do next.

To zoom back out before diving back in later – our ray tracer takes parameters like image size, samples per pixel, scene behavior, a selection of which scene that we've created to render – in our demo, dynamic user information affecting the camera's position – recursion depth, and number of threads. It outputs an image of the rendered scene and timing information.

Data structures include bounding boxes (Bounding Volume Hierarchy, abbreviated BVH henceforth) to optimize memory accesses to the world and the world information itself. In our implementation, that's spheres, planes, and their materials, which we store in objects in a world vector. These data structures are repeatedly being probed by our camera renderer via millions of projected rays – which is computationally expensive and is the main target of parallelization in this project. We were able to take a data-parallel approach to this in our MPI implementation by instantiating multiple copies of the worlds amongst various processors, and having each one compute a less computationally intense image, before combining results together through communication. Finally, we struggled with locality due to the divergent behavior of rays in our scene, but locality is generally exhibited in nearby regions of pixels. While we did not attempt an SIMD implementation, we believe this would've been ineffective due to this divergent behavior.

BVH:
BVH is a data structure that aims to minimize the number of memory accesses needed to track down the location of an object. The basic idea entails bounding a number of primitives in a cube/bounding object. If a ray misses the bounding object, we know it misses all objects enclosed, saving us from checking them individually. *Without BVH we have to check every single object for every single ray to see if we have a collision*. Bounding objects are hierarchical and approximate a binary search when looking for an object. Whenever we add an object to the world, we insert it into the tightest bounded object that already encloses it, or created a new bounding object to enclose it. We experimented with implementing a BVH, which will come up later in the report.

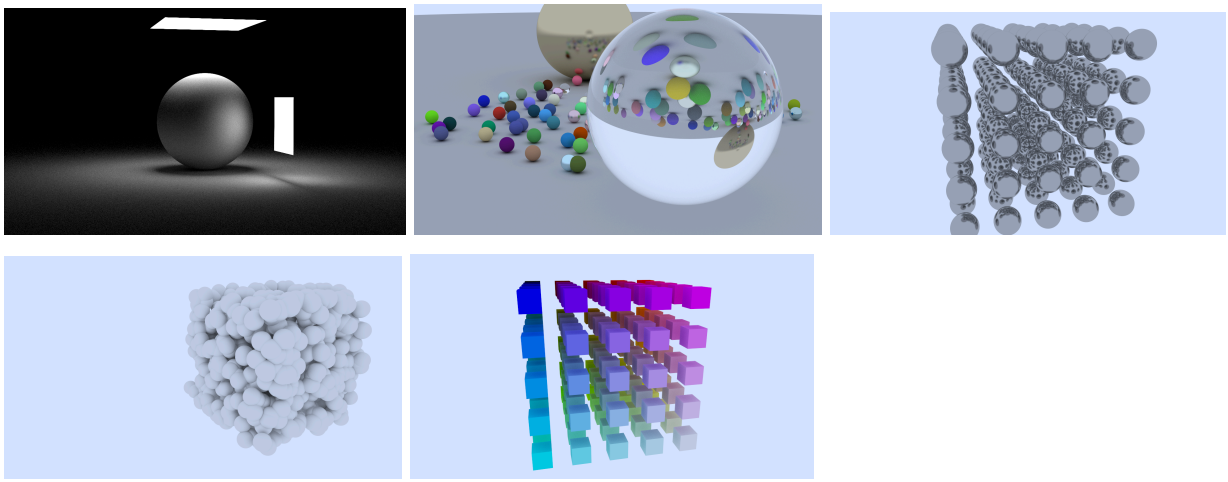
Approach

As neither of us knew about raytracing or computer vision before this project, we began by learning about it and following the tutorial from the books "Ray Tracing In One Weekend", and "Ray Tracing: the Next Week" by Peter Shirley, Trevor David Black, and Steve Hollasch. This is what we modeled our original, C++ sequential implementation off of and expanded upon.

We began by first implementing the rendering system outlined in the first book, then selectively explored elements like quadrilaterals, BVH, and lighting from the latter parts of the series, bringing these into our sequential version. We created five scenes to benchmark against (further explored in Results):

- simple: a simple scene of two spheres, but with two quadrilateral lights
- materials: a more dynamic scene with 104 sphere primitives showcasing the three light-scattering materials we implemented, metals, lambertian surfaces, and dielectric. Objects were partially positioned behind a large dielectric (think glass) sphere as a way to introduce more complex regions of computation in the scene to promote work imbalance
- grid: a grid of dynamic size of uniformly colored metal spheres arranged in a cube
- biglittle: one thousand uniformly colored lambertian sphere primitives arranged within a single cube that takes up only one region of the final image. We were inspired by the "biglittle" scene from the CUDA circle renderer homework for this, which we created to introduce more work imbalance
- color_grid: a grid of dynamic size of multicolored lambertian cubes arranged in a larger cube, to explore the effects of color and the quadrilateral primitive more deeply

Scenes are shown below, ordered left to right, top to bottom, as introduced above



Then, we created a camera system that would allow us to render rotating images and how many frames to render in a complete rotation, which we believed could help even out workload, as well as introduce new behavior for rays within a scene (i.e, for `materials` we have fewer rays that go through our the dielectric sphere as the camera rotates, for `biglittle` we show the cube in different regions of the image). And finally, a timing system for all of this, a way to easily change program behavior via command line arguments, and an SDL demo that allowed a user to control the camera with their keyboard and render scenes in real time (although this was relatively useless without parallelization due to speed).

OpenMP:

OpenMP was the first, and theoretically, should've been the easiest, implementation we started with. However, we ran into numerous problems that ultimately led us to completely rebuild much of our original sequential code.

In all implementations, the rendering code follows the basic pseudocode:

```
Unset
# render the image
for j in image_height
  for i in image_width
    pixel_color = black
    for _ in the number of samples_per_pixel
      initialize a ray r randomly concentrated around projection of pixel (i,j)
      pixel_color += ray_color(r, max_depth, world)
    image[(i,j)] = average pixel color in samples

# get the color contribution of the ray in the scene
function ray_color(r, depth, world)
  if we exceeded the maximum depth
    return black
  if ray r did not hit anything in the world (world + objects access)
    return camera background color
  color_from_emission = emitted color of material (object + material access)
  ray scattered = scatter the ray according to material (material access)
  color_from_scatter = attenuation * ray_color(scattered, depth-1, world)
  return color_from_emission + color_from_scatter
```

This gives us three immediate candidates for OpenMP parallelization: around the outer loop over image columns, the inner loop over image rows, or the innermost loop over samples. However, we ran into two main challenges in this approach: first, and unknown at the time to us, but the original library we used for randomness was not thread safe. Each call to randomness, which was used to randomly place the original ray, and to update the new location of the scattered ray recursively, used the same generator, updating its seeds between threads and causing significant artifactual communication. This was more easily remedied by using the older, but thread-safe random double generator `drand48()`. Secondly, we ran into significant challenges in memory access between the world in our for loop, which includes all the objects in the world and the materials of those objects, which were all instantiated as a series of classes that we had to "chase pointers" to access. An additional issue we had to track down later, was the duplicated inclusion of multiple header files between threads, which we were able to remedy with the `#pragma once` header, and using header guards.

Originally, with 8 threads OMP was actually slower to render our testing scene. I'm looking at my notes for this, and it was a difference of 14.24 seconds for 1 thread, 38.98 seconds for 8 threads, or around 2.76x *slower*. We also had 624,418 cache misses for 1 thread, 4,036,467 cache misses for 8 threads, around 6.46x more cache misses, which was the first indicator of artifactual communication.

While attempting to track down these issues, I hypothesized that the issue was due to the shared world between processes. I created a thread-based implementation (not using OpenMP) to explore this further, and found that even when each thread created its own world and camera, we had similarly poor results in terms of speeding down with more threads, and about a linear increase in cache misses.

I continued by removing the BVH hierarchy that we added to the original code, and was able to get a speedup of 2.2x with 8 threads – still poor, but immediately we can identify the memory accesses of this shared, global object causing poor performance, where calls to this object were made even if we had separate world objects to store our primitives.

Examining the code more I realized we had additional issues in the shared objects that our code relied upon to compute things like vector normals, collisions, and compute the way material collisions affected rays. Furthermore, I researched the shared pointers that our original code was using and found that smart pointers were hurting our speedup as each thread had to maintain its own counts for how many times the pointed object was used in the scene, which bounced around threads when being updated. I restructured our code to omit these pointers, and to be less "object-oriented", reducing the number of pointer chases we would have to complete to locate objects and made our code reliant on fewer objects. For example, rather than using a shared object to compute and do math on intervals, I reorganized the code so that logic would be computed inline or with functions instead. This was the first big series of improvements, and gave us a speedup of around 3.9x with 8 threads using OMP, compared to 2.2x.

I researched OMP pitfalls online and was able to identify randomness and a lack of header guards as another potential issue. Using a thread-safe randomness algorithm in the camera gave us a speedup of 3.5x with 8 threads, and then extending this to our material objects, increased speedup to 5.54x.

Finally, to minimize workload imbalance, dynamic scheduling increased speedup to 7.02x on this test scene. This speedup, the best with 8 threads we had at this point, placed the pragma header on the outermost loop which allowed OMP to dynamically schedule threads to work within all pixels, and performed moderately better than at the inner loops. I'll explore this in results, but the overhead incurred by dynamic scheduling actually hurt our speedup for our most simple scenes, but proved incredibly useful for more complicated scenes with more primitives. However, dynamic scheduling allows OMP to better allocate resources to an inherently imbalanced problem, where different pixels can have vastly different complexities depending on how many objects their rays project onto, and their materials/further recursive interactions. It was at this point that we collected and analyzed our first results in bulk for OMP (outlined in Results).

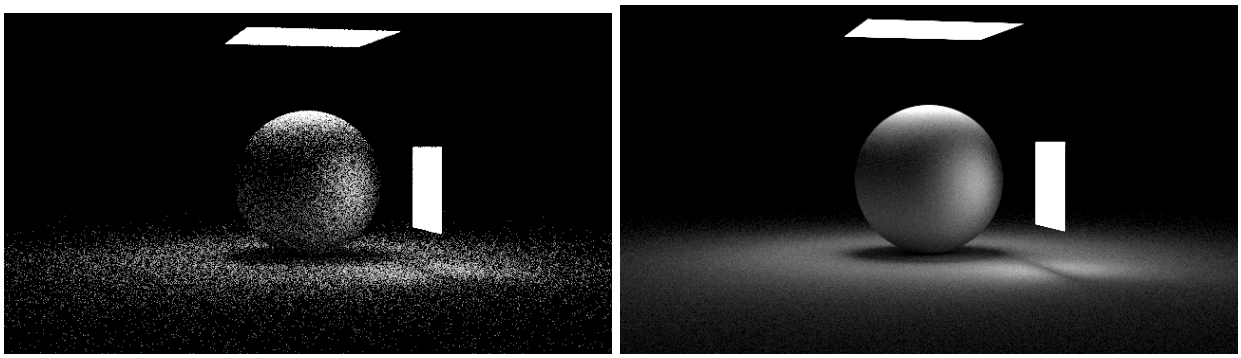
Later experiments with OMP involved modifying our code so that objects scattered light less often in order to reduce divergent behavior. Recall in our pseudocode the line `` `color_from_scatter = attenuation * ray_color(scattered, depth-1, world)`. `` For lambertian surfaces, the original code scatters all rays and attenuates by the color of the material. However, it is also possible to scatter rays with only a fixed probability p, and attenuate by $color/p$. This reduces how deep we have to recurse for each pixel (reducing divergence, ideally improving performance), but produces less accurate images. Results are explored in a later section.

Furthermore, I circled back to BVH and successfully reimplemented the structure in the new code style – not relying on shared objects to do all of the computations – and was able to successfully get a speedup with BVH enabled (results explored later).

MPI:

MPI was the second parallelization method I sought to implement, after changing the code to properly implement OMP. The first attempt involved a main thread building up the world and communicating that information to workers on the fly, but this involved creating many new MPI datatypes and slowdowns due to communication. The second method, which ended up being much easier to implement, was a data-parallel approach where each processor created its own world according to the chosen scene, processed its own user inputs, and then rendered a less computationally complex version of the world with its own randomness by taking fewer samples than necessary. In the end, processors communicated their computed images to a main thread, which then averaged the colors to produce a final image. This had the effect of moving the innermost for loop in the pseudo code, and "outsourcing" it to the various processors.

When scaling to more processors, I reduced communication between processors by having them combine their final images hierarchically. Rather than the ROOT process gathering images from workers, each worker paired up with another worker, combined their image, then repeated then in rounds so that any image could be combined in log(nproc) steps, each step requiring communicating only one image between processors, greatly reducing the footprint of communication and stress on the root processor.



The above shows an example of what one of these undersampled images would look like at first (this is what one of 64 threads produces), then what the final image looks like once all nproc of these images are combined.

Results

From the approach section, recall that results were computed by analyzing the renderer's performance on five different scenes, and by varying various inputs to the renderer.

TODO:

- OMP had better speedup on GHC machines, even better speedup with BVH
- Image size didn't really do anything
- Explain size experiment - with OMP, pretty uniform results, once we surpass a minimum amount
- Scatter - doesn't really affect speedup but does affect compute time

OMP and MPI Results

We computed both Computed Time and Total Time using the `chrono` library– where Total Time includes the time it takes to do things like process user inputs, initialize OMP/MPI, generate the world, and write images, while Computation Time is just the time it takes to create the image in the scene (including communication time for MPI). Only Computed Time is analyzed below, but for MPI results are slightly less effective if we factor in total time (it takes longer to initialize MPI).

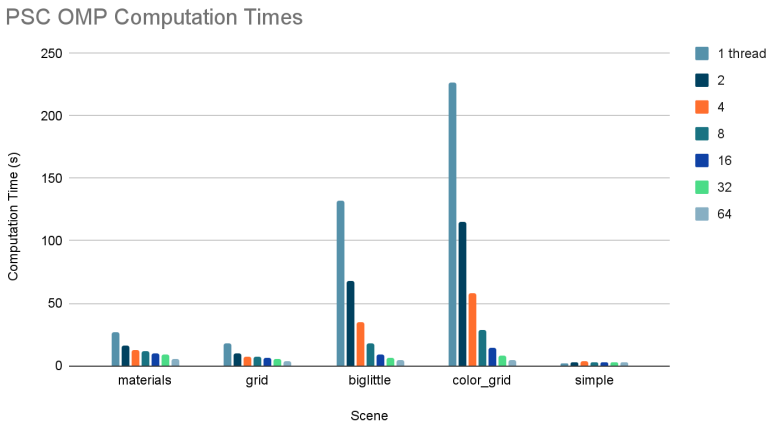
Unless otherwise specified, the following specifications were used:

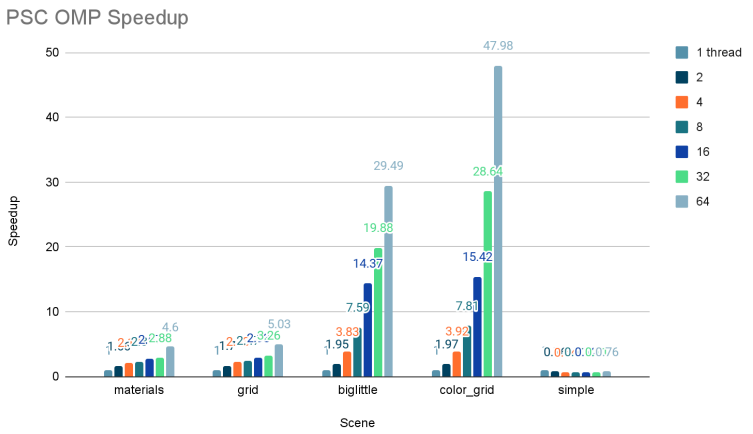
```
Unset
image size: 640x360 px
samples per pixel: 64
max ray recursion depth: 50
use BVH: no
animation: none
number of primitives in grid and color_grid scenes: 5^3=125
```

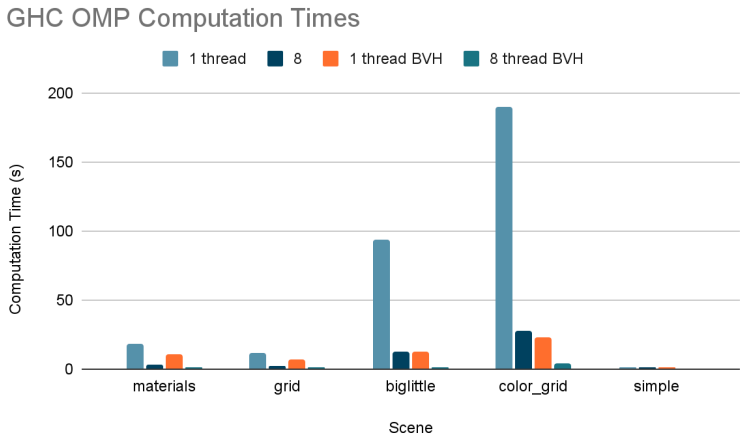
Before diving in – I found that MPI had the best scaling and speedup due to it's well load-balanced implementation. In that implementation, each worker had its own world, and computed the same image, all pixels in it, which gave us the best load balancing and memory access. In ethos it's very similar to Data-Parallel methods we saw for things like LLM training, complete with ring communication at the end to combine results. OMP had limited speedup on the PSC and did not scale as well – although had close to linear speedup on the GHC machines.

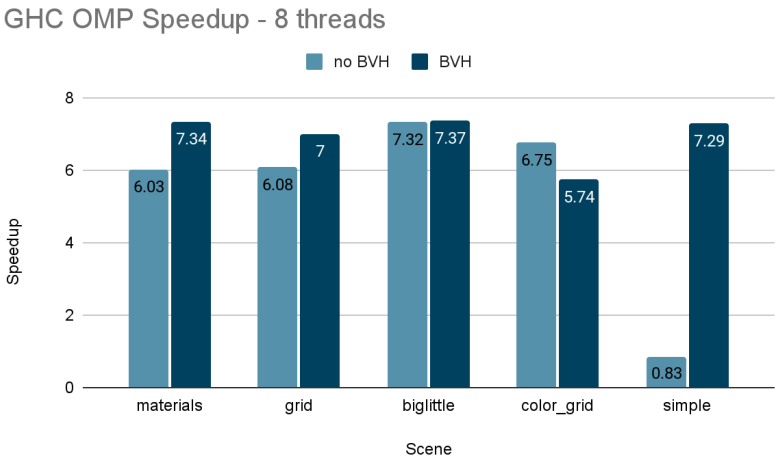
BVH did not scale as well as I would've initially expected on the PSC, although this could likely be an issue in being bandwidth-limited. BVH did however lower the computation time for all scenes wherever used, although had the greatest improvement on scenes with a high number of primitives, as we would expect. For OMP, for all but one scene – `color_grid` – BVH improved speedup on the GHC machines (although strangely, this is not true for the PSC OMP implementation, but this can likely be explained due to differences in CPU and memory capabilities).

Speedup scaling on GHC and PSC Machines, and BVH









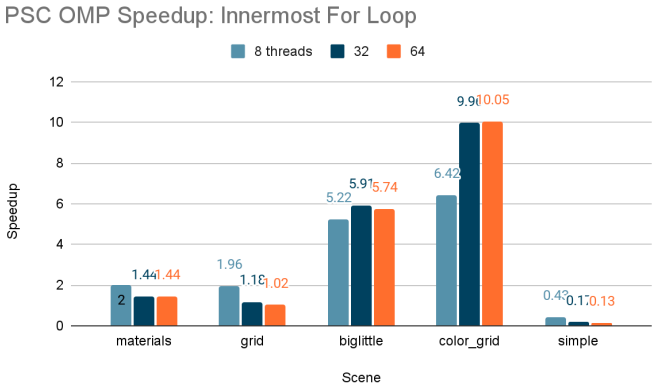
Above, we can see that OMP has a maximum speedup of 47.98x on the `color_grid` scene with 64 cores, while having a quite limited speedup on smaller scenes like `materials` (4.6x with 64 cores). For scenes with few primitives like `simple`, we actually get fractional speedup. On the GHC machines, we get a better speedup for 8 threads that's pretty close to linear, for all scenes except `simple`.

We see the best speedup on complicated scenes that either have a high number of primitives (like `color_grid` and `biglittle`). These scenes are very difficult to render sequentially, but in parallel become much faster. Looking at the cache misses (see below), we see that the arithmetic mean of cache misses to threads is lower in parallel, showing that the parallel version exhibits better locality, improving performance. This makes sense as without BVH, the renderer has to loop through *every single* object in the scene for each ray. If we do this in parallel, for nearby regions of the image, we likely have the relevant objects already in the cache. This can also explain why we got worse speedup with BVH enabled on a scene like `color_grid`. Looking at the

computation times, the 1 thread BVH renderer happened to be significantly more efficient, meaning that the benefit due to locality scaled less with 8 threads than without BVH.

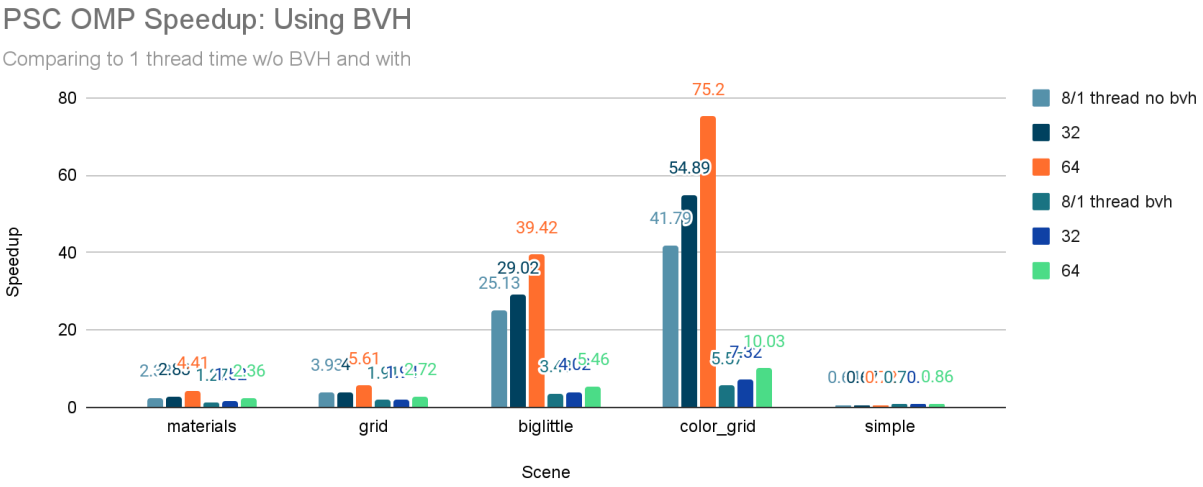
Comparatively, for scenes with few primitives like `simple`, the overhead in OMP scheduling is likely partially responsible for this fractional speedup – switching to static scheduling improves speedup from 0.82x to 0.89x. BVH also makes lighting in a scene like `simple` more efficient to access in memory, which is likely the reason for the more linear speedup with BVH enabled.

To address the scaling issues, seeing that MPI had better speedup as the number of threads scaled (results below), I repeated the PSC experiment again, this time parallelizing over the number of samples in a pragma reduction rather than over the pixels. I hypothesized expecting that this would scale better, even if less efficient for smaller threads, as hopefully it would lead to a more balanced workload. However, this wasn't the case, and we only get worse, more uniform speedup:

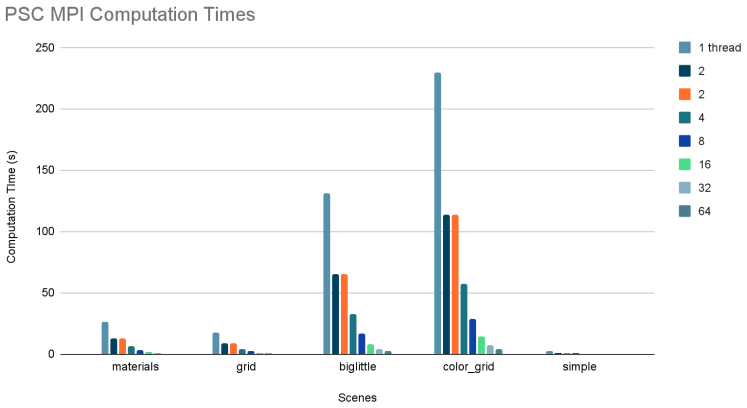


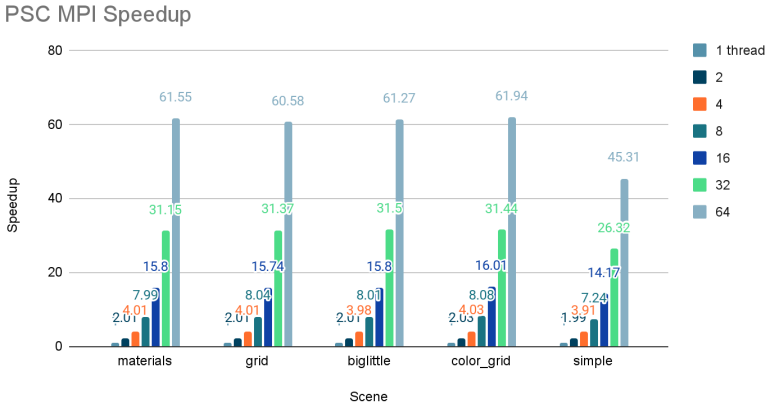
As our speedup remains similar at 32 and 64 threads for even scenes that scaled much better before, I believe then that this approach is therefore likely memory limited in terms of accessing the world, which the MPI implementation would avoid in terms of having multiple copies of the world created for each worker.

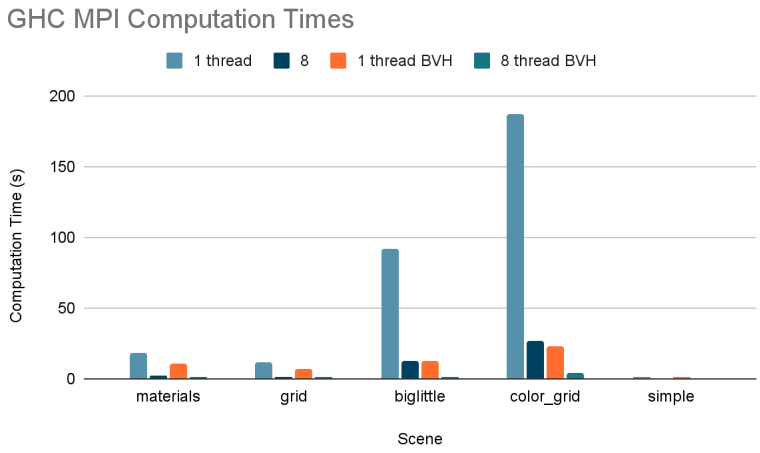
These findings are further suggested when we look at the speedup when using BVH. In the graph below, the first three columns represents the speedup of the parallel versions when compared to the sequential version that *doesn't* use BVH. The next three columns are the "more fair" versions that compare speedup to the sequential version that does use BVH, for which we can see worse speedup. So while BVH improved speedup on the GHC machines for all scenes except `color_grid`, it made speedup *worse* on the PSC machine, and scaled poorly. Differences in the GHC vs PSC processors' abilities to access memory likely explain this (more analysis below in Cache misses section).

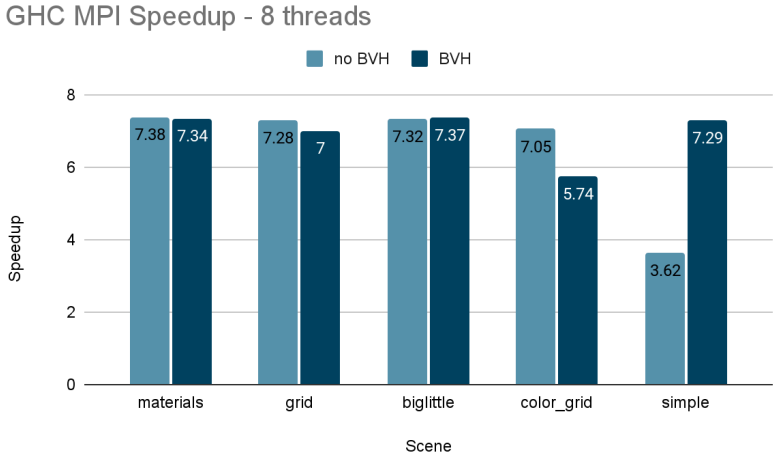

PSC OMP Speedup: Using BVH
Comparing to 1 thread time w/o BVH and with
Speedup
Scene
materials
grid
biglittle
color_grid
simple
8/1 thread no bvh
32
64
8/1 thread bvh
32
64
75.2
54.89
41.79
39.42
29.02
25.13
10.03
5.46
5.61
4.41
2.36
0.86

MPI speedup:


PSC MPI Computation Times
Computation Time (s)
Scenes
materials
grid
biglittle
color_grid
simple
1 thread
2
2
4
8
16
32
64


PSC MPI Speedup
Speedup
Scene
materials
grid
biglittle
color_grid
simple
1 thread
2
4
8
16
32
64
61.55
60.58
61.27
61.94
45.31
31.15
31.37
31.5
31.44
26.32
15.8
15.74
15.8
16.01
14.17
7.99
8.03
8.01
8.08
7.24
4.01
4.01
3.98
4.03
3.91
2.01
2.01
2.01
2.03
1.99


GHC MPI Computation Times
1 thread
8
1 thread BVH
8 thread BVH
Computation Time (s)
Scene
materials
grid
biglittle
color_grid
simple


GHC MPI Speedup - 8 threads
no BVH
BVH
Speedup
Scene
materials
grid
biglittle
color_grid
simple
7.38
7.34
7.28
7
7.32
7.37
7.05
5.74
3.62
7.29

MPI speedup is significantly more uniform as a result of more uniform load balancing and less artifactual communication in memory accesses. Between threads, workload distribution can only vary by the randomness in each sample, but with millions of rays being computed this deviance evens out.

We see a worse speedup for `simple` – likely because the overhead in communication for combining images with 8 threads dominates computation time. With BVH we see a better speedup – but, note that with only three primitives BVH introduces unnecessary overhead in a scene. This increases computation time in the 1 thread case, meaning that the communication time is comparatively less of an overhead in the 8 thread case.

Cache misses (BVH enabled)

OMP:

Scene	Cache misses (1 thread)	Cache misses (8 threads)	Ratio (8 threads / 1 threads)
materials	152,176	149,233	0.98x
grid	109,680	143,732	1.31x
biglittle	112,496	139,091	1.24x
color_grid	114,214	152,370	1.33x
simple	97,510	143,052	1.46x

MPI:

Scene	Cache misses (1 thread)	Cache misses (8 threads)	Ratio (8 threads / 1 threads)
materials	2,689,459	7,274,910	2.71x
grid	2,658,577	7,296,714	2.74x
biglittle	2,663,433	7,213,286	2.71x
color_grid	2,663,459	7,192,062	2.70x
simple	2,661,880	7,419,473	2.79x

Cache misses (No BVH)

OMP:

Scene	GHC Cache misses (1 thread)	GHC Cache misses (8 threads)	GHC Ratio (8 threads / 1 threads)	PSC Cache misses (1 thread)	PSC Cache misses (8 thread)	PSC Ratio
materials	153,810	155,777	1.01x	43929	42483084	967.09x
grid	116,773	143,776	1.23x	30471	25328175	831.22x
biglittle	184,751	220,041	1.19x	53227	26469607	497.30x
color_grid	274,156	261,200	0.95x	901774	24131098	26.76x
simple	97,191	144,731	1.48x	25681	10030963	390.60x

MPI:

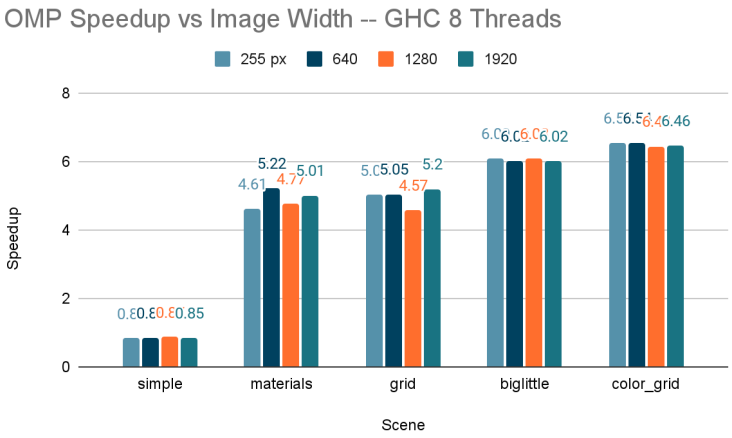
Scene	GHC Cache misses (1 thread)	GHC Cache misses (8 threads)	GHC Ratio (8 threads / 1 threads)	PSC Cache misses (1 thread)	PSC Cache misses (8 thread)	PSC Ratio
materials	2,689,525	7,062,444	2.63x	2179697	15842847	7.27x
grid	2,655,217	7,346,477	2.77x	1809321	15510423	8.57x
biglittle	2,816,656	7,185,452	2.55x	1800252	15702214	8.72x
color_grid	2,871,089	7,612,656	2.65x	2039122	16558578	8.12x
simple	2,637,821	7,191,595	2.73x	1674730	15273535	9.11x

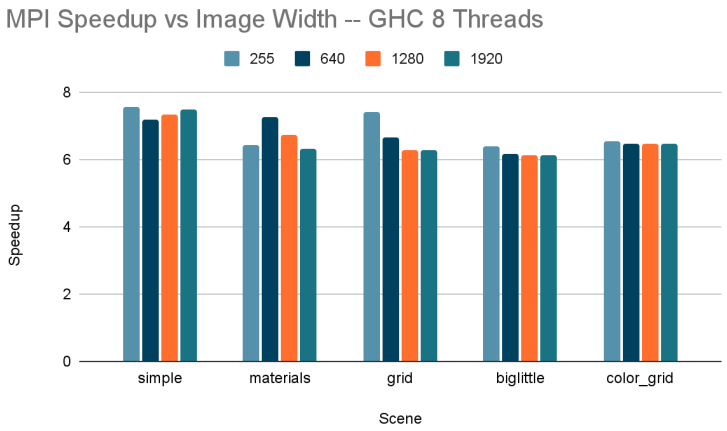
If ever there was data to explain the differences in performance and scaling between GHC and PSC, this is it! This is the smoking gun.

For OMP, on the GHC machines, with 8 threads, we see roughly equal or only slightly higher number of cache misses compared to the 1 thread renderer. However, on the PSC machine, this *skyrockets*, we see *hundreds* of times more cache misses. It seems then that the PSC cores are just less efficient at memory access than the GHC machines, or have smaller L1/L2 cache sizes. For MPI – which notably scales significantly better on PSC, but has comparative results on GHC – we see around 2-3x times more cache misses with 8 threads on GHC, and an expected, linear 7-9x times more cache misses with 8 threads on PSC. This certainly has to explain the differences in scaling and the conflicting results we saw on the different machines.

Affect of Image Size on Speedup

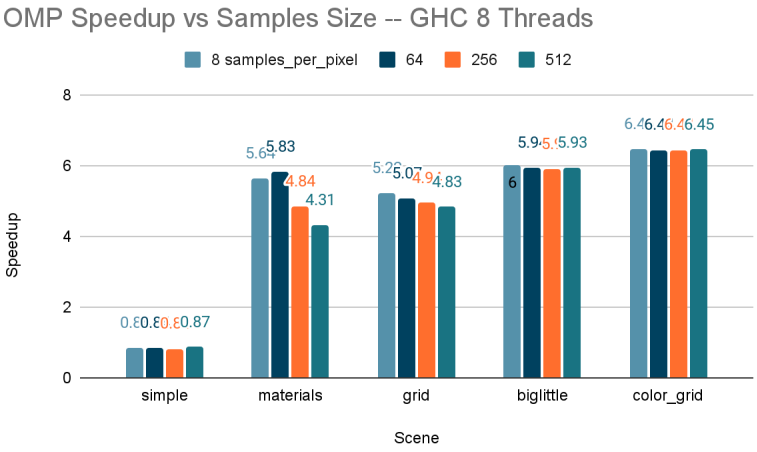
Predominantly I did not find a significant difference between the speedup due to image size for OpenMP or for MPI. The only clear pattern is that speedups for scenes that take more total time to render, like `biglittle` or `color_grid`, are more uniform, which suggests that differences in speedups earlier are due to more random fluctuations in scenes that don't take as long to render.

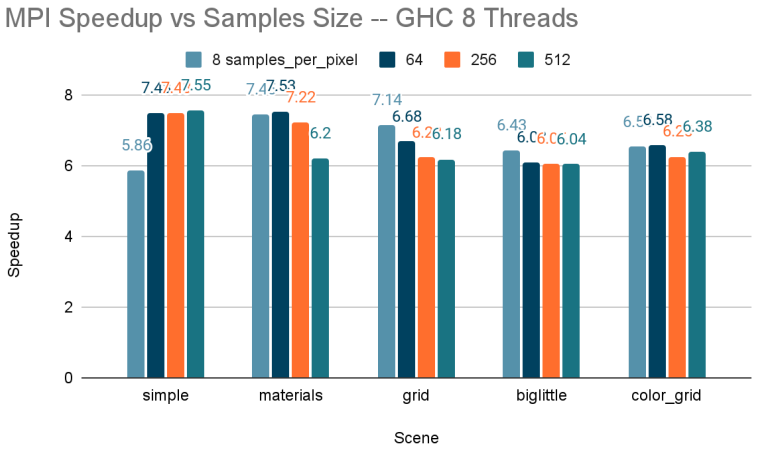




Affect of Sample Size on Speedup

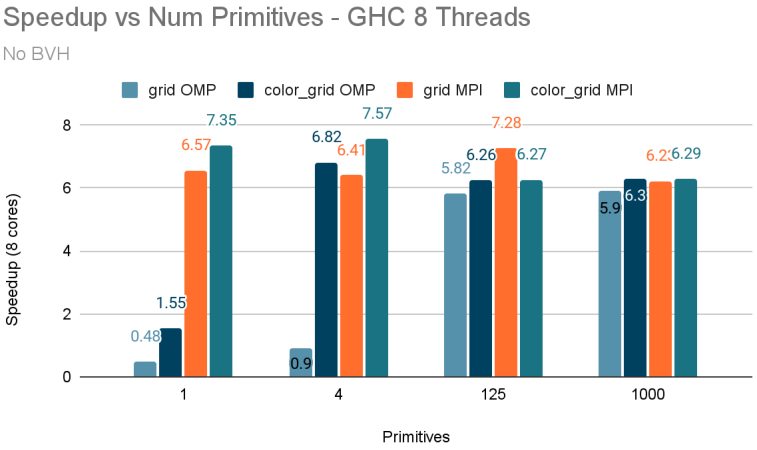
Sample size closely follows the patterns found in image size above. This makes sense, as increasing the total number of pixel and the number of samples per pixel both have the same effect of uniformly increasing the number of rays that need to be computed.

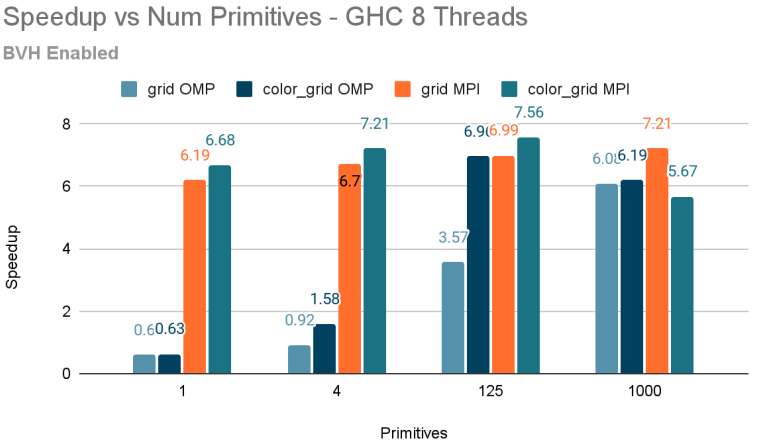




Affect of Number of Primitives and Speedup

Recall again that BVH introduces a marginal overhead that should make rendering scenes with large primitives more effective, while the cost of this overhead is more expensive for scenes with fewer primitives. The graphs below then show this behavior, where on small scenes BVH doesn't scale as well in parallel, as it doesn't justify the overhead it incurs compared to its benefit.

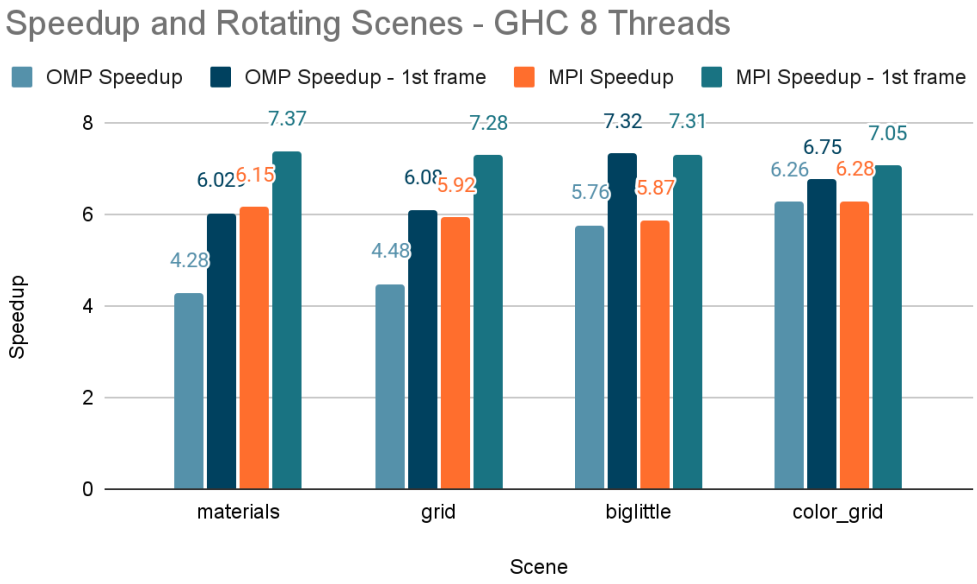




For OMP we sometimes see a worse speedup with BVH enabled – for example, color_grid on one thousand primitives. This is likely because it speeds up the 1 thread version so much, that the bottleneck in parallelization shifts from computing rays, to communication costs between more threads, which arguably is a good problem to have if it lowers our computation time either way. Also note that internally, boxes (in `color_grid`) are implemented as 6 plane primitives to create 1 box, which perhaps shows why these findings are more prominent for `color_grid` than it is for `grid`, where one sphere is one primitive.

Affect of Rotation on Speedup

For the following experiment, I generated 30 frames of rotation for each scene, where the camera rotated 2pi/30 radians around the center point each frame, and measured the speedup over the course of the entire process. As the angle of the camera changes, the objects may either get closer or further away, and we view then from different perspectives, leading to more data about each scene being collected.

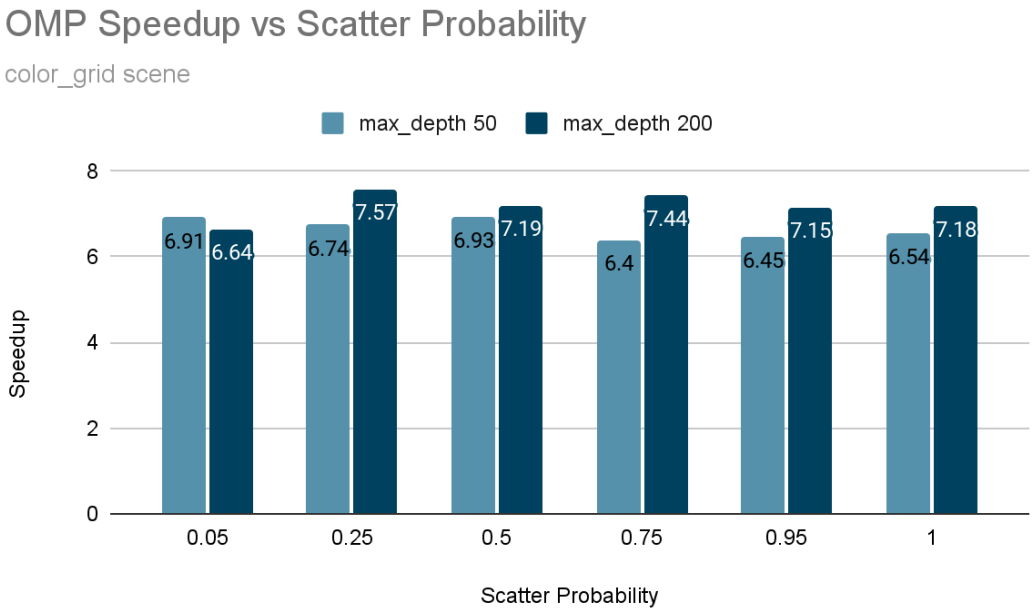


Here, rotation was slower for both OMP and MPI by about the same amount, which suggests that the scenes are perhaps in more favorable initial positions in terms of speedup for their first frames.

Affect of Sample Probability on Speedup and Correctness

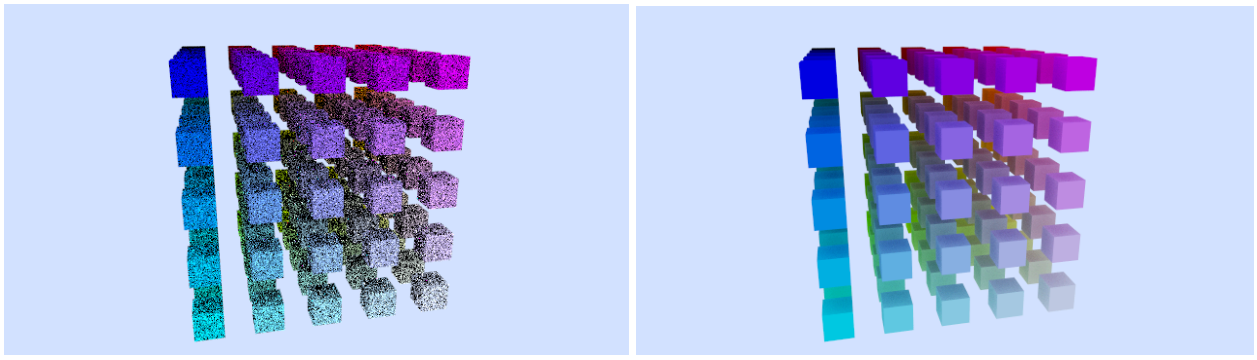
To expand on something mentioned in the Approach section – every time we hit an object, we call it's material's `scatter` function to see what color it contributes to the ray, if we should a scatter a new ray after that, and if so, how that ray should be scattered. For the implementation of Lambertian surfaces, we always scatter a ray and attenuate (modify the color of the relevant pixel) by the object's color, or internally, `albedo`. However, it is also possible to scatter with only a fixed probability `p`, and then attenuate by `albedo/p`.

This experiment attempted to determine the effect of scattering with various values for `p`. The original expectation was that if we scattered *less*, then we would recurse less far into each ray and thus have less divergent code, and therefore a better speedup with OMP. Results are presented for the scene `color_grid` below (which had the most lambertian surfaces), varying the maximum recursion depth possible between 50 and 200.



The results are somewhat confirmed – the hypotheses would be that 5% scatter prob would have the highest speedup since it has the least divergent behavior in terms of rays. However, this ended up being 25% scatter probability instead. Furthermore, it looked like 100% – always scatter – was more effective than scattering 95% of the time. This also makes sense, where with 95% so close to 100% it's not likely to significantly affect how deep we recurse with our rays, but it does introduce a new need to generate randomness and another conditional branch to go through, limiting speedup.

However, it's also important to note that even if lower scatter probabilities tend to have better speedups, the images they produce are noisier as they contain less information. We compared results to reference images, and found that 5% scatter probability, the image deviated by 4.8% from the reference, while at 50% scatter probability this was reduced to a 1.03% deviance. The between 5% scattering and 100% scattering is shown below:



References

As stated and linked above, we referred to the “Ray Tracing In One Weekend” series to start our sequential code version.

Work Distribution

Cassidy:

- Created Github repo to host website and project website
- Created Project Proposal and Project Milestone Report
- Contributed OpenMP / MPI Results to Poster Session
- Created demo for poster session
- Create test cases, scenes, testing harness
- Adapted and built upon “In One Weekend” sequential code to create our sequential code the sequential code I benchmarked against in the project
- Implemented, optimized, and benchmarked OpenMP and MPI implementation
- Wrote final report sections: summary, background, approach sections for OpenMP and MPI, results sections for OpenMP and MPI

Akintayo:

- Contributed to Project Proposal and Project Milestone
- Contributed CUDA Results and Background to Poster Session