SUMMARY:
In our project, we used different parallelization techniques to improve on different aspects of ray tracing. We then applied these parallelization techniques to a wide variety of different scenes with varying difficulty levels measured in terms of scene materials, mediums, and number of objects. Based on the results measured, we delve into explanations for how these relatively performance gains arose.

Background:
In computer graphics, we often want to be able to render physically accurate images through some mathematical description of the scenes. To truly be able to gather the physically accurate color and luminance of each point on the plane is difficult, as we would need to accumulate light contributions from an infinite amount of angles coming from an infinite amount of directions. Instead, we estimate the color and luminance through Monte Carlo integration, where we take a large amount of random sample directions and sum up their contributions based on how much light from that direction contributes to the light on that pixel on our canvas.

This is how ray tracing works in practice: we shoot out a ray through the center of each pixel of the screen and calculate the color of that pixel by accumulating colors from the albedos. Then, as the ray hits an object in the scene, we again draw a random direction, and continue tracing the ray with this new direction. This operation is not too expensive by itself, but notice that in order to trace even one pixel we already had to have drawn at least two random samples, calculated the collision against objects in the scene, recalculated the ray directions, and then recursively repeated these steps. The amount of times we have to repeat these operations, both recursively and across all pixels on the canvas, makes the computation time grow exceptionally fast. Firstly, there can be hundreds of thousands of pixels on a canvas. A standard 1080x1920 pixel screen has over 2 million pixels, each of which will require a raytracing operation to fill in. On top of that, taking only one sample per pixel will result in a horribly noisy image. Since raytracing relies on Monte Carlo integration, we must take a large amount of samples per pixel in order for the pixel color to converge towards the true value. Even then, we might get very noisy images based on the behavior of the scene. In particular, specular surfaces are difficult to render due to their property of reflecting light in very specific ways.

Given all of this, ray tracing benefits greatly from any sort of acceleration, as each little thing we optimize will give significant speedup due to how many times the operations are repeated. More importantly, each raytracing operation is independent from one another. This leads to ray tracing being an easy target for parallelism, *theoretically*. However, as we have realized through this project, due to the fundamental nature of ray tracing and its inherent independence, that many parallelization techniques don't perform too well on it. This will be expanded upon more in the "Challenges" section.

What are the key data structures?
In ray tracing, there are a few important primitives and crucial acceleration structures.

- Triangles: triangles are the backbone of most complex objects. Large objects are decomposed into triangles because many operations are easy to compute on triangles, and we don't want to have to redevelop our algorithm to fit every shape imaginable. Some operations that triangles must support are find barycentric coordinates, which gives the weighting of each of the vertices on a center point in the triangle.
- Spheres: spheres are used commonly because having only one primitive can be limiting at times. Spheres are great for demonstrating certain material properties, such as glass or specular surfaces, and allows us to be able to create round edges without having to mimic it with hundreds of tiny triangles.
- Bounding Volume/ Box Hierarchy: similar to quad trees, BVHs are used to speedup ray tracing operations. In the summary above we mentioned collision checks against objects in the scenes. For any realistic scene that you can commonly find in movies and video games, we would be required to do collision checks against tens to hundreds of thousands of objects due to the triangle decompositions explained above. Instead of wasting huge amounts of computational resources and checking every object, we instead partition objects into subsets based on their positions. These subsets are then divided further until some requirements on their numbers are met. We then draw bounding boxes around these subsets, and then bounding boxes around multiple bounding boxes, and so on and so forth until our largest single bounding box now encapsulates every single object. Then, when we check for collisions, we can first check if the ray hits the bounding boxes. If not, then we can skip all the objects in that bounding box, thus speeding up computation.

What are the key operations on these data structures?
Given what is mentioned above, we need to be able to quickly perform collision checks against the primitives. Thus, we need to be able to compute distances, find normals, and compute collision points on the primitives. For the BVH, we need to be able to insert, possibly update or delete if we are processing moving scenes (not in the scope of this project), get bounding box, and check collisions against boxing box.

What are the algorithm's inputs and outputs?
The inputs to the program are scene files that describe the scene objects, where they lie, their material properties, textures, and meta datas of the scene including what kind of sampling technique to use, how many samples to take per pixel, which integrator to use, etc.

The output of the program is an physically accurate (as much as we can make it) image that is the result of ray tracing the scene described.

What is the part that is computationally expensive and could benefit from parallelization?
As described above, the part that is computationally expensive is the sheer amount of samples we have to draw, the object collision calculations, and the new direction calculations, spread over layers and layers of recursive calls.

The Challenges:
Without spoiling our results too much, while we did achieve significant speedup, we were not able to hit linear speedup across any of the parallelization techniques we tried. To truly understand why this is the case, we will now take a deeper dive into ray tracing and present a fundamental clash of design principles between ray tracing and parallelization.

1) Diverging paths: ray tracing at a high level just has a simple triple for loop, looping through the width and height of the image and then through every sample per pixel. These top level for loops are simple to parallelize. However, as we get deeper into each recursive call, the parallelizable parts start to become nonexistent. The fact that the rays diverge from each other, hit different objects, pass different branch checks, then bounce a different number of times before fizzling, already severely hampers SIMD approaches to parallelization. On top of that, each single ray tracing operation have inherently little to no parallelizable work, as it only draws one (or a very small, negligible amount) of samples, does collision checks against up to two BVH boxes at once, and then recurses. Each single ray's operations are simple, short, and require no help on its own. It is only when we have a huge amount of these operations that the program becomes slow.
2) Fundamental conflict: you might be asking, what if we turned the program around, and instead of doing "for each pixel, send a ray to check collisions and accumulate color", we do "for each object in the scene, figure out which pixels they collide with, and then add color to it"? If we do something like this, then SIMD parallelizations become far easier. We can check parallelly, for every shape, every pixel it covers, and then sum up all the colors. However, what is described here is known as rasterization, a fundamentally complementing technique to ray tracing. By simply changing these for loops to range over objects in the scene, we completely change the behavior or our program and ray tracing techniques are no longer applicable. Thus, this further complicates our chances of finding a good SIMD implementation.

Break down the workload. Where are the dependencies in the program? How much parallelism is there? Is it data-parallel? Where is the locality? Is it amenable to SIMD execution?
As mentioned many times already, the workload is that we need to repeat a simple operation hundreds of thousands of times. Across different ray tracing operations, the program is completely parallelizable, as they have no dependencies. However, doing so won't always achieve optimal speedup, as some ray directions will interact with far more objects, thus leading

to slowdowns in collision checks, or have far stronger contributions to the overall color, which will lead to far more recursive calls. Between each recursive iteration of ray tracing, the operations are also independent, and thus can be parallelized. However, since we generated the starting points of the rays using the ending points of the previous ray, we won't know where to send the next recursive call before the current call completes first. There *is* a way around this: through bidirectional path tracing, we can generate random directions on both the initial camera ray's hit location and on points in the scene, after which we can connect them together using some probabilistic estimates and sum up their colors. Then, each of these ray generations can be parallelized. However, bidirectional path tracing is an advanced topic that is difficult to implement correctly on its own, and thus we did not further explore this possibility.

APPROACH: Tell us how your implementation works. Your description should be sufficiently detailed to provide the course staff a basic understanding of your approach. Again, it might be very useful to include a figure here illustrating components of the system and/or their mapping to parallel hardware.

OpenMP:
The OpenMP implementation is the simplest of our three techniques. Since each ray is independent, simply calling "parallel for" already achieves a large speedup of about 5-6x. Depending on where the "parallel for" is put, we can get slightly faster or slower results. The fastest results came from putting the "parallel for" at the top level, for each column of the image. Putting the pragma at the top level (per row) or bottom level (per pixel sample) result in slower times. This is because the chunksize division is too large in the first case and too tiny in the second case. In the first case, we have a lot of wasted threads that will sit around and do no work as some contiguous rows are going to take significantly longer than others, thus we suffer from wasted resources. In the second case, the assignment is too fine grained and each thread ended up completing their work fast enough that they are constantly contending over the next tasks and thus incurring major overhead.

This is proven using the top command. In the first case, the CPU utilization starts out at 800% as expected, but starts to drop to 700% at around 60% completion and then to about 400% at around 90% completion. This is because while some threads are still doing their work, others have already finished. Changing between static, dynamic, and guided schedulings do not do much to help alleviate this. In the second case, utilization is nearly always capped out at 800%, but the runtime is slower.

There are many reasons why OpenMP doesn't achieve a perfect 8x speedup. One reason is that we chose a sample size of 100 but a chunksize of a multiple of two, thus the last chunk had leftovers and we had to wait for it to catch up and complete. Another reason is that light contributions are stronger or weaker depending on the ray's directions. If the ray's contribution is

stronger, we do not prune it as early and let it recurse more. This will cause some rays to take significantly longer than others to complete (at least 32x in some of the scenes). These hypotheses are confirmed with the “top” command, where occasionally the CPU utilizations will drop to 700-750% then go back up, which corresponds to some threads finishing their work, and utilization will drop to around 150-200% at around 90% completion, which corresponds to the last chunk of work being leftover.

OpenMPI:
The OpenMPI implementation was one of the more involved techniques that required extremely rigidly structured communication between the parallel machines and more careful planning in deciding how to balance the workload between the mentioned parallel machines. Parallelism through a message passing model requires highly structured data in the form of messages and OpenMPI is not an anomaly. The main implementation detail is that the parallel machines split up the work in computing the overall color for each pixel in the image. This is because each pixel’s final color can fundamentally be independently calculated without referencing another pixel. This means that splitting up the work can be condensed down to each processing understanding which pixels it would have to compute for. The amount of work per pixel, however, is not uniform and it becomes a challenge in understanding how to balance it for each parallel machine. The amount of computation required per pixel depends on the surface and the number of rays that end up affecting the pixel and the load balancing technique is discussed in detail later on. The main load balancing technique was to allocate more resources to the more dense areas of the image which would command more ray interactions.

The OpenMPI implementation targets a system with multiple cores very well because when it executes on a n-core system, it can interact with all n cores as separate processors. The original code had to be changed to be mapped to be a more shared memory focused environment. The original serial code had some data protected in C++ private class attributes, but since we want each processor to be able to access data from the same data structure, such as the data in the image class, that data was pulled out of the private attributes and made public.

It was definitely not a straightforward path in implementing parallelism with OpenMPI. Even the very first step was taken in a direction completely different from the final implementation. The initial rendition tried to send the tree representation of the surfaces in the image so that each processor could independently process the surfaces and report to each other the values in the trees. This in theory would allow each tree to have an extremely easy time computing the final image values as they would just have to increment based on tree lookups instead of recursively testing each ray interaction. However, this plan fell short in implementation when we ran into hurdles in understanding how to send this data structure and how to send different sized data structures efficiently. Even if we created a new MPI data structure to forward and broadcasted sizes so each processor would be able to call Allgatherv, the idea fell short when considering

cases where some trees would have over 500,000 nodes such as in the ajax-ao test case. Instead, the final implementation involves splitting across the image itself, where each processor takes on a separate part of the image. There were three main methods of implementation that were tried which included splitting the image by rows, columns, and in a grid-like fashion. In addition to being able to avoid sending large quantities of data, splitting in this fashion also introduces the possibility of exploiting locality since the same ray will likely affect nearby pixels. The first implemented version splits on the rows and provides each processor with an equal number of rows to compute on. The next was an obvious choice, repeating the same strategy but on the columns instead. The last split employed was a square split in a grid-like fashion which can try to exploit the locality found in the rows and the columns.

Load balancing for the OpenMPI implementation is a difficult task where each pixel is computed independently and only once. This ultimately rules out options that dynamically allocate the pixels to threads based on the time and computational intensity of the previous run. The main decision for load balancing was to distribute the workload based on the density of the surfaces used in each rendered image. Each image is described by a json file that includes all of the surfaces and their expected positions on the rendered image. The surfaces are parsed and an overall bounding box is created that encompasses the resulting final locations of the surfaces. The load balanced version uses this as a metric of the density or concentration of where most of the computation will be because that is where most of the interactions with the rays will lie. The pixels outside of this box are then considered of less computation intensity and one processor would be able to efficiently compute those pixels. This essentially directs more processors to a subsection of the image that is more dense so that they can focus in on a denser area with more computation. The load balancer does not perform in the same manner when the bounding box is of a similar size to the entire image or as a very smaller size of the entire image because we don't want to create more imbalance when trying to reduce the imbalance. When the bounding box is too large, one processor ends up only computing the values outside of the bounding box, which leaves the other processors with more work. When the bounding box is too small, it could be the case that there were not enough interactions where most of the processors would end up finishing too early and remain idle. The load balancer goes to a default split where n-1 processors focus on 3/4 of the image.

SIMD

As explained by the Challenges section above, SIMD parallelizations are very difficult to adapt to for ray tracing. In this section, we only parallelized the sample generation. Instead of generating samples when the rays collide and bounce, we generate all samples at the beginning using SIMD instructions, save them to an array, and simply reference them as needed. Given that sample generation takes far less time than collision checks, we only achieved negligible speedup with SIMD. The upside to this is that with SIMD we can perform many sample generations at

once. However, there is a downside to having to access the array memories and possibly kicking out cache lines for object collisions.

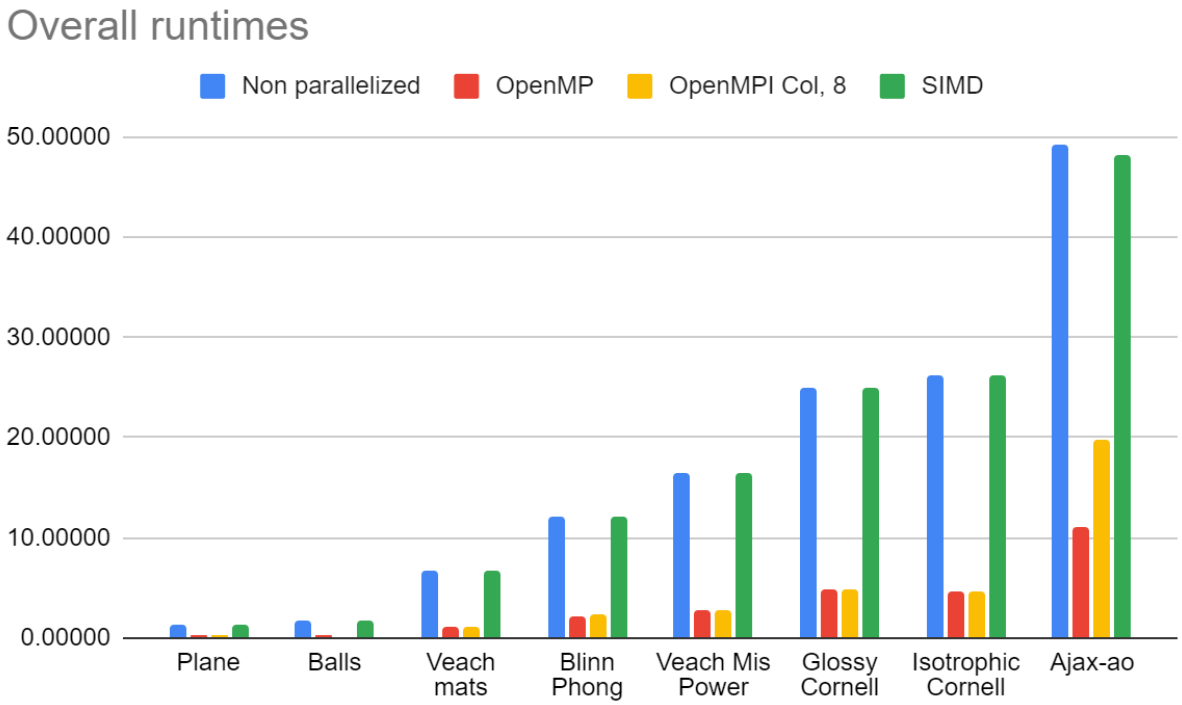
Describe the technologies used. What language/APIs? What machines did you target?
We used C++ along with OpenMP, OpenMPI, and Intel ISPC targeting the Gates machines.

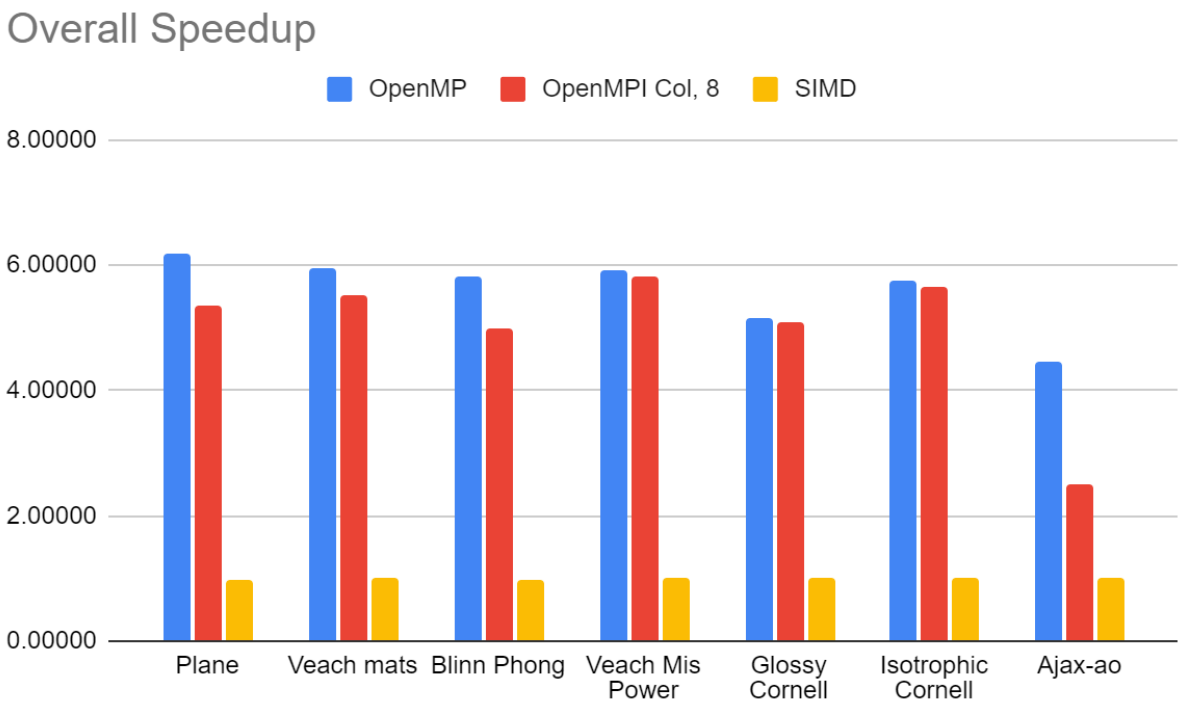
• Describe how you mapped the problem to your target parallel machine(s). IMPORTANT: How do the data structures and operations you described in part 2 map to machine concepts like cores and threads. (or warps, thread blocks, gangs, etc.)
Described in detail above, essentially each bundle of ray is mapped to a thread block and ran.

• If your project involved many iterations of optimization, please describe this process as well. What did you try that did not work? How did you arrive at your solution? The notes you've been writing throughout your project should be helpful here. Convince us you worked hard to arrive at a good solution.
The process has been covered earlier, but the fundamental idea was to try different methods for each parallelism model and compare the performance for each.

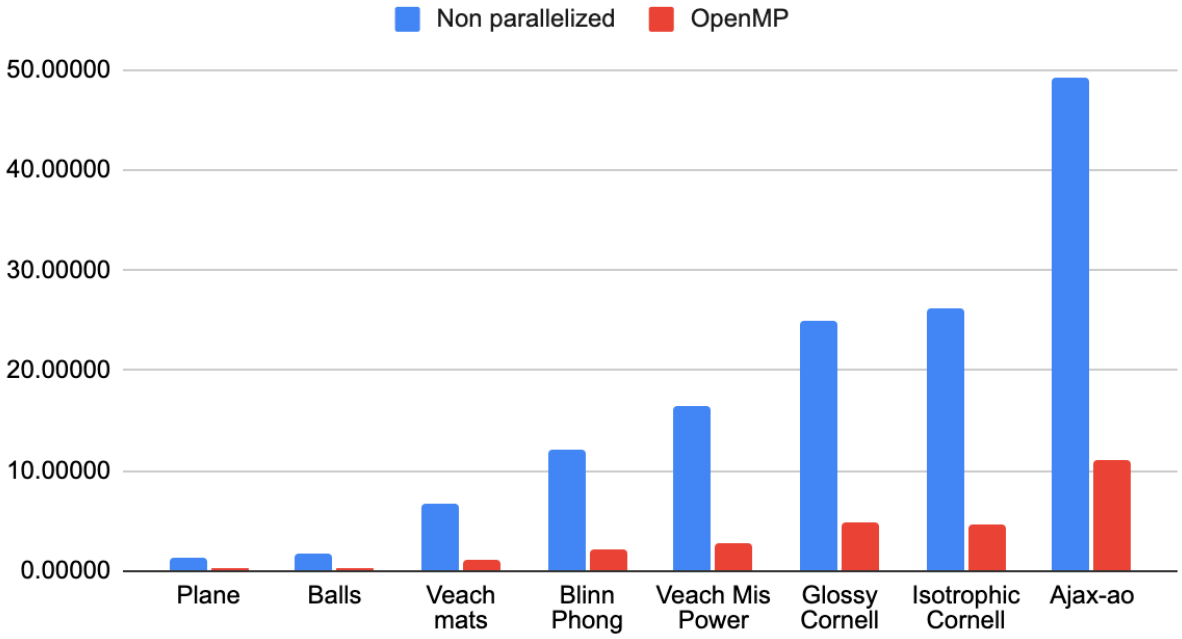
• If you started with an existing piece of code, please mention it (and where it came from) here.
We used code from 15-668 Physics Based Rendering. The code itself was part starter code, part final project code written by Siheng Li.

RESULTS (All time measured in minutes):

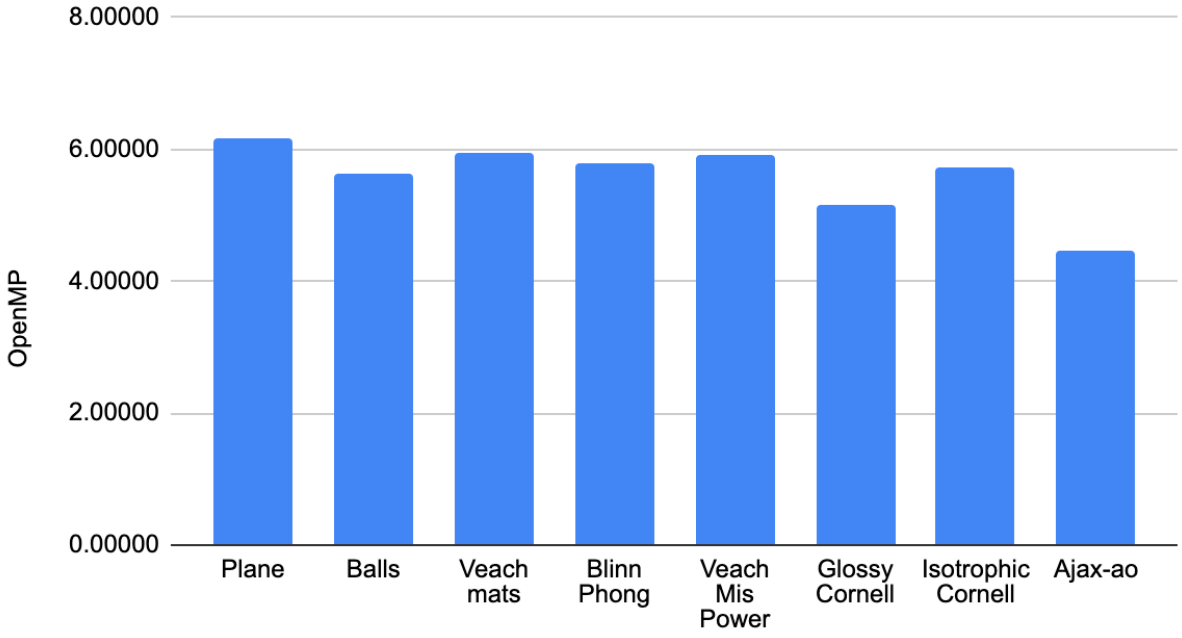




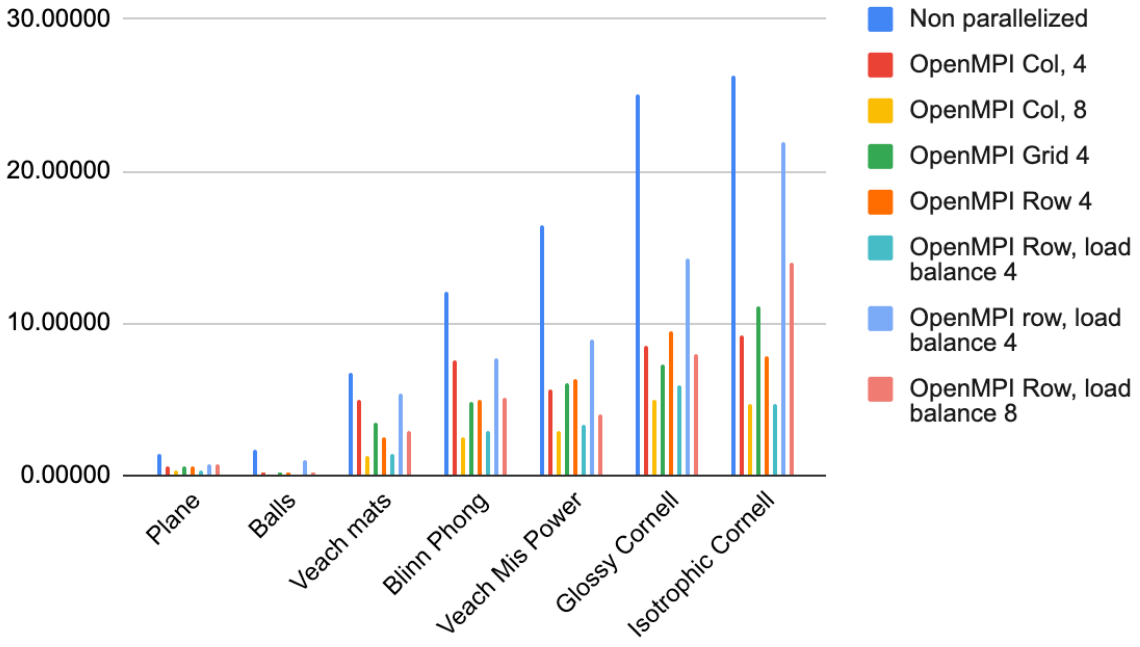
OpenMP runtimes



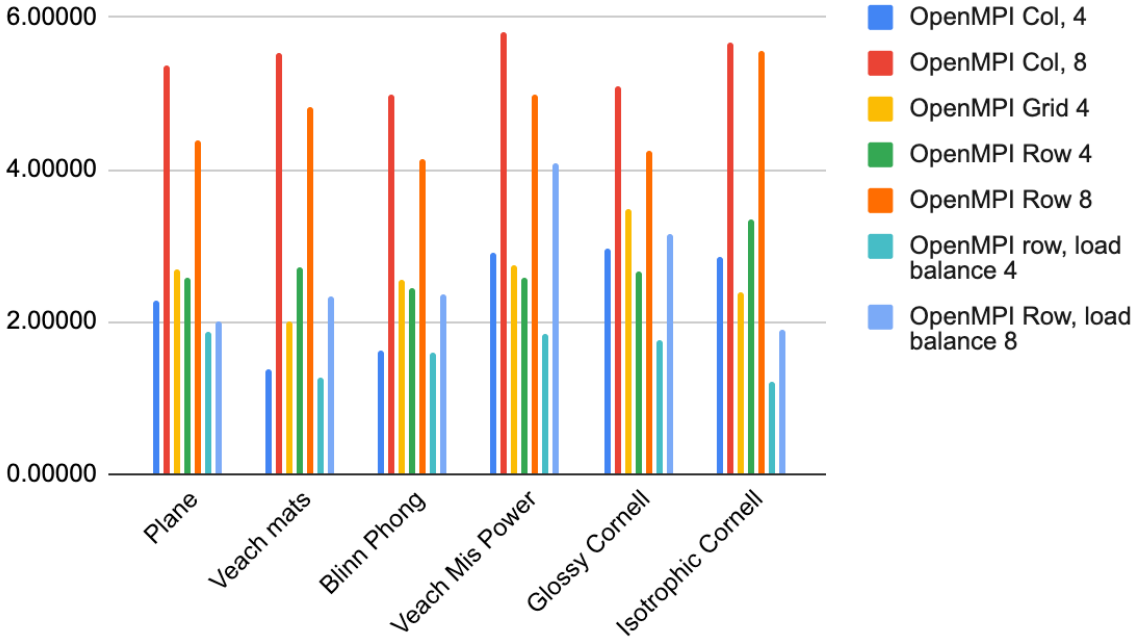
OpenMP Speedup

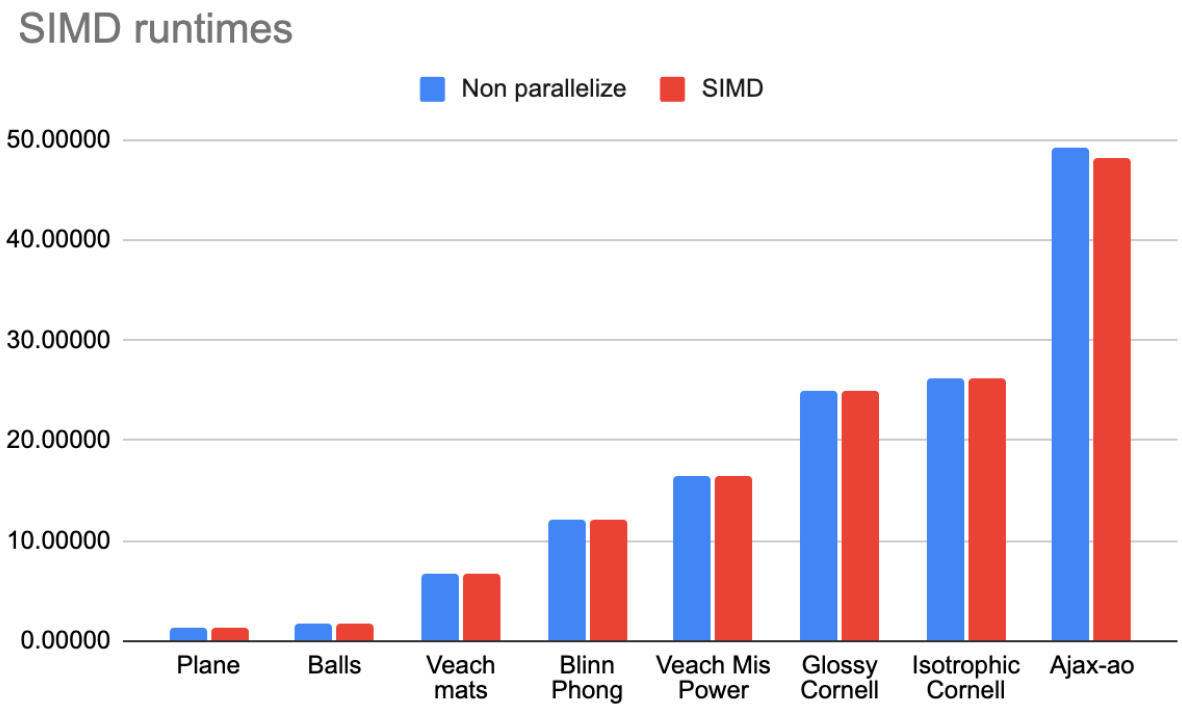


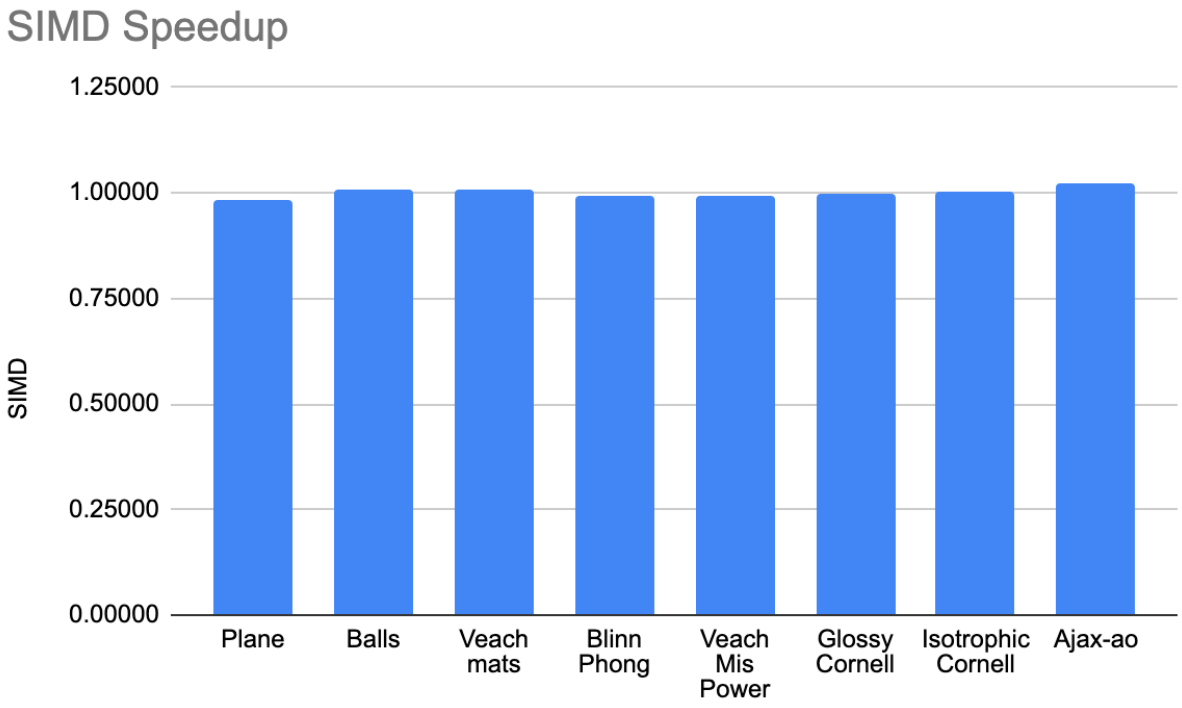
OpenMPI Runtimes



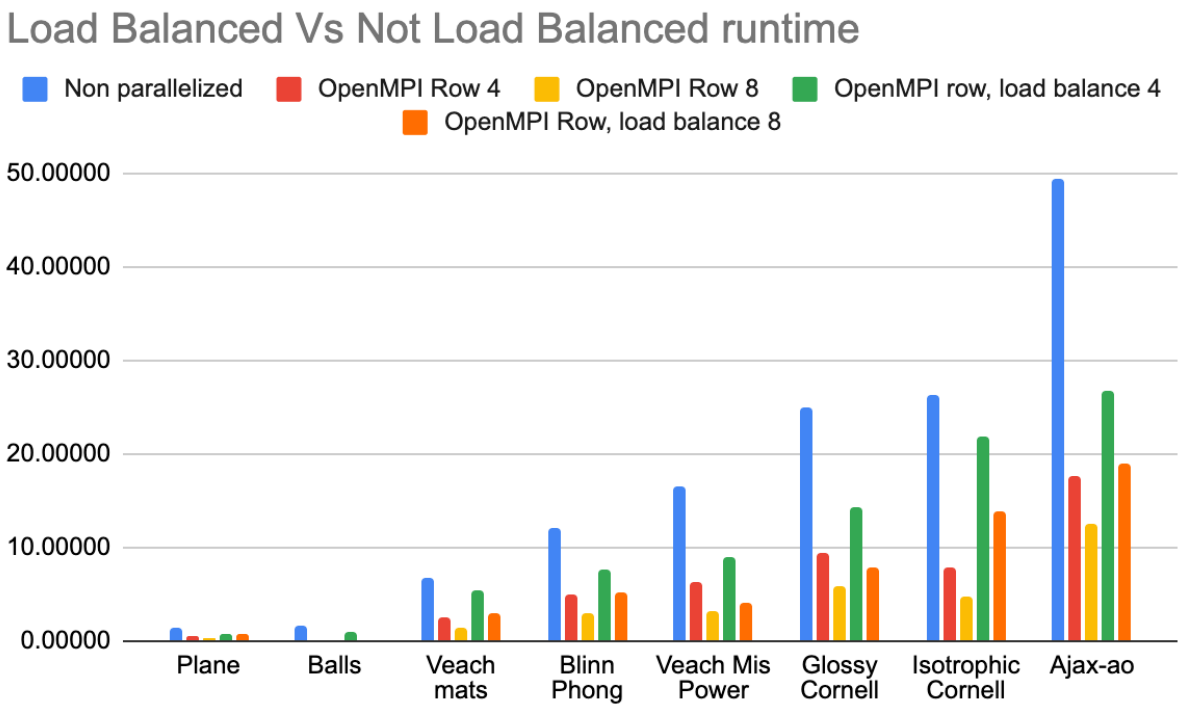
OpenMPI Speedup

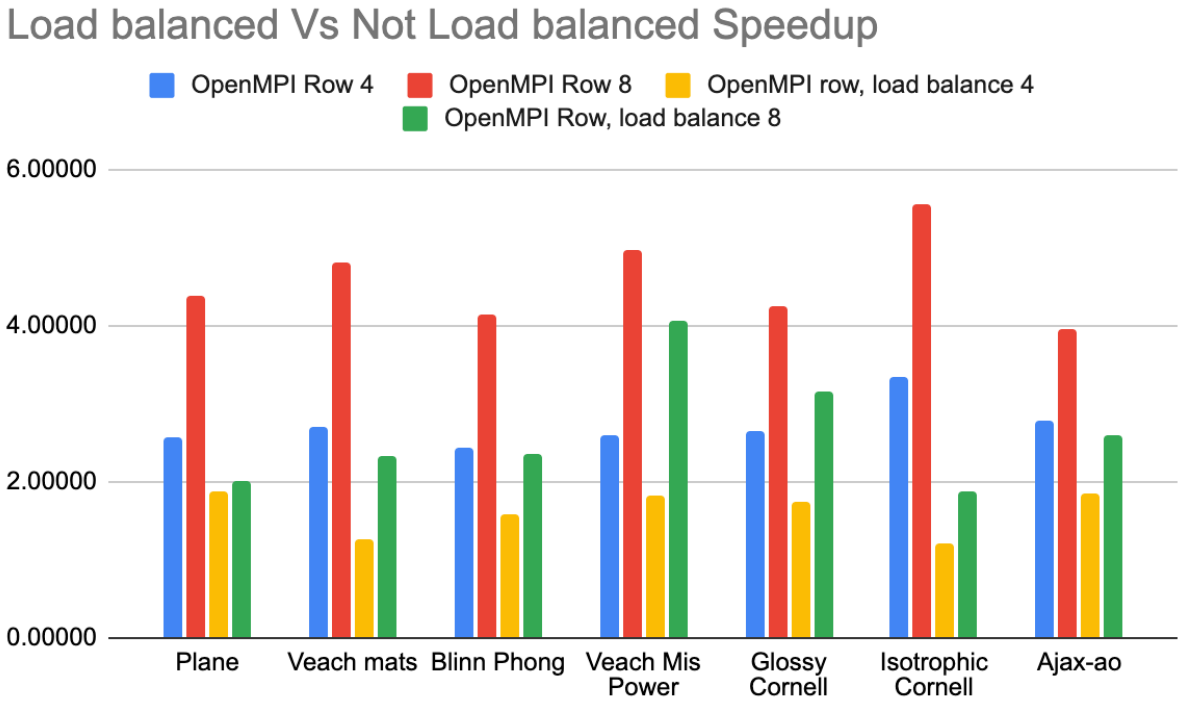






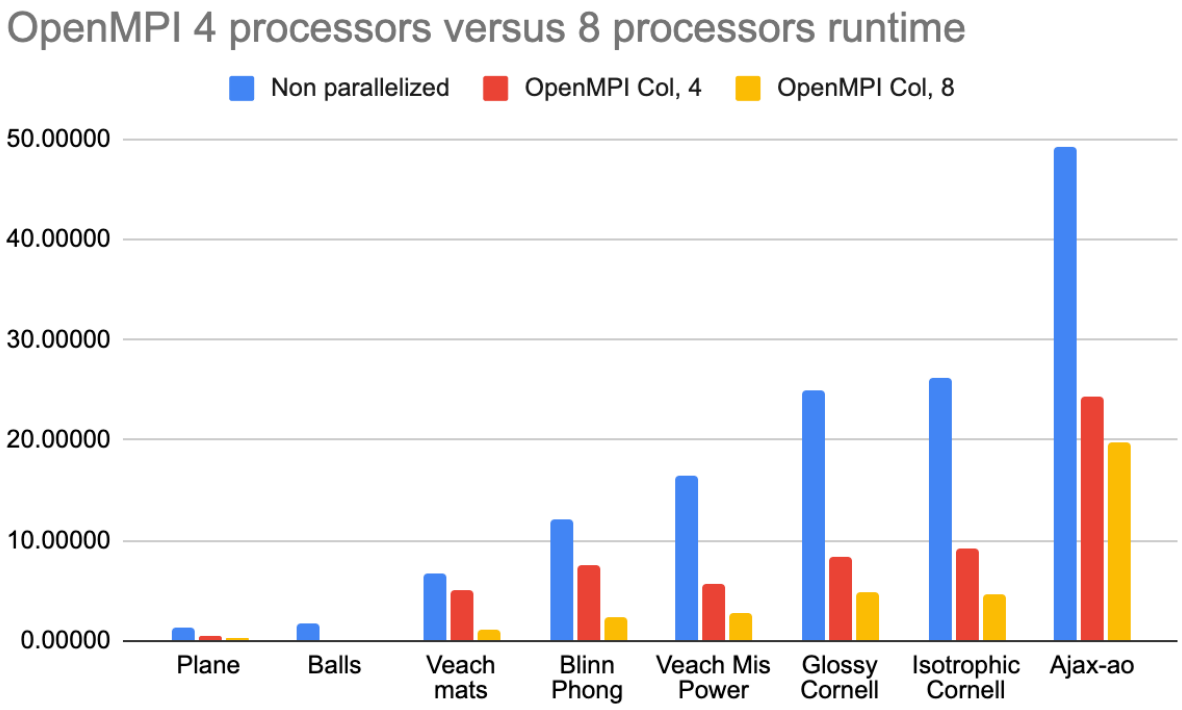
From our results, we can conclude that OpenMP was able to provide the best speedup with OpenMPI coming in at a close second. It is fundamentally difficult to achieve speedup with SIMD here because each pixel's calculation requires differing quantities of data.

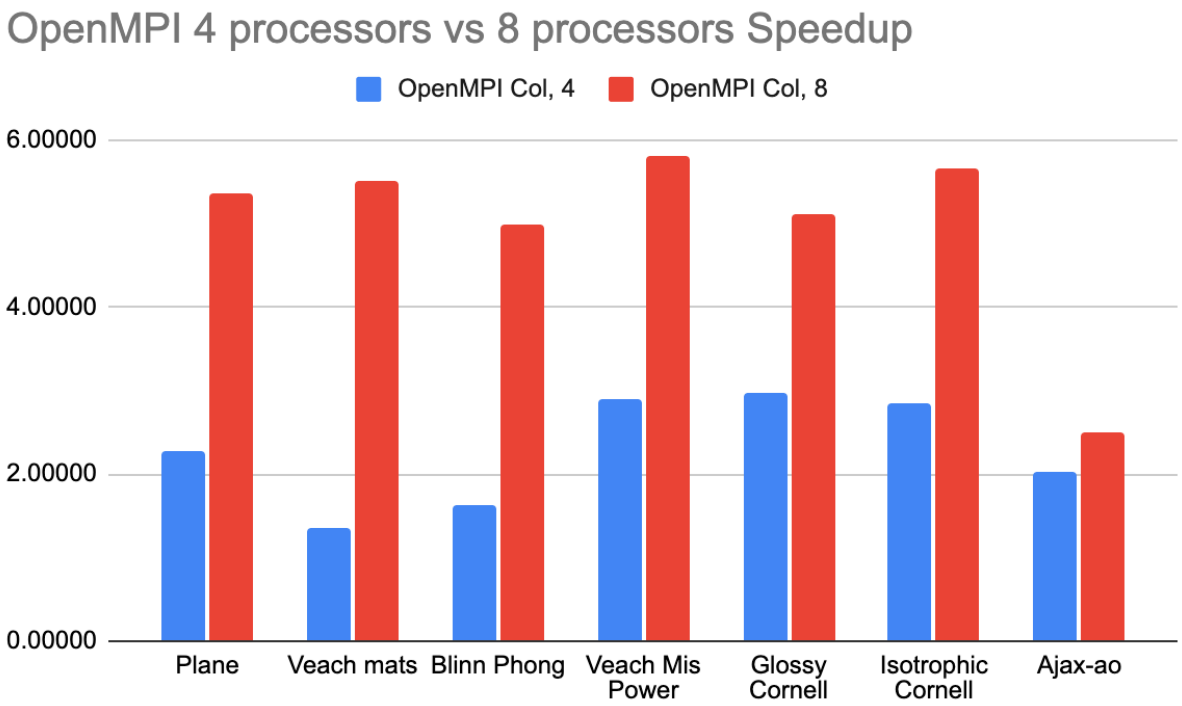

Load Balanced Vs Not Load Balanced runtime
Non parallelized
OpenMPI Row 4
OpenMPI Row 8
OpenMPI row, load balance 4
OpenMPI Row, load balance 8
50.00000
40.00000
30.00000
20.00000
10.00000
0.00000
Plane
Balls
Veach mats
Blinn Phong
Veach Mis Power
Glossy Cornell
Isotrophic Cornell
Ajax-ao


Load balanced Vs Not Load balanced Speedup
OpenMPI Row 4
OpenMPI Row 8
OpenMPI row, load balance 4
OpenMPI Row, load balance 8
6.00000
4.00000
2.00000
0.00000
Plane
Veach mats
Blinn Phong
Veach Mis Power
Glossy Cornell
Isotrophic Cornell
Ajax-ao

The load balance technique here was able to balance the workload amongst the processors, but it also incurred some overhead cost that could be improved upon. There is another metric reported that is not represented by the graphs that is reported by the program when it finishes executing in the average number of intersection tests per ray. For example, when running the regular version with 4 processors, the averages are 4.1, 2.2, 2.0, and 4.7. When running with the load balanced version, the averages were 2.6, 3.7, 2.8, and 3.5. This indicates that the workload was split more evenly, but there was too much prologue and epilogue costs to see a performance increase. The load balancing could also be improved on in predicting the number of ray interactions and allocating in a more systematic way. It is to be noted that it is fundamentally hard to predict the number of ray interactions before recursively identifying all of the dependencies.

The decision to load balance on the row implementation was decided on preliminary results that we were not able to reproduce. The data was gathered preliminarily to produce a baseline on which implementation would be the most fruitful to load balance on. After collecting a substantial amount of data, it would make more sense to load balance the column version because it was actually higher performant, but this was noticed during the analysis without enough time to implement another version. The results with the load balanced row version still provide insight, just a little less performant.





We also wanted to explore the overall scalability of the parallelism, but the gates machines are limited in processor counts available for MPI. With more time and less concern about the availability of resources, we would have also liked to have tried using more processors on PSC. We are conscious that each render is very computationally intensive and takes a large amount of time, so we ultimately did not test on PSC, but we can still analyze the speedup observed when we run on 4 processors versus running with 8 processors. Most of the rendered images experienced around a 2x speedup when doubling the processors. This is a good indicator that increasing the problem size doesn't reduce the effectiveness of the introduced parallelism. It is important to note that there are cases such as Ajax-ao where there are pixels that run in very long dependent chains that make it difficult to parallelize. Ultimately the parallel scheme scales well with the number of processors, but not with scenes that have long dependent chains.

• **IMPORTANT**: What limited your speedup?
As mentioned in detail above, the presence of dependencies and constant divergence limited our speedup.

Deeper analysis:
The major components of the code are collision checks, generating new ray directions, and drawing new samples. The collision detection takes significantly longer than the other two operations. We have to check against up to two bounding boxes and then recurse on their children. In order to achieve more speedup, we could perhaps let each leaf bounding box have a higher number of children and then perform collision checks against them parallelly.

Was your choice of machine target sound? (If you chose a GPU, would a CPU have been a better choice? Or vice versa.)
We chose to run the code on the CPU because the Dirt renderer that 15-668 uses is a CPU based renderer. The code could have been similarly ran on the GPU, but the choice of CPU vs GPU is not significant here because in order to parallelize on the GPU without the threads in the warp diverging and thus causing significant slowdown, we would need to change the recursion and pruning, which is essentially what the CPU implementation also require.

REFERENCES:
Code by `Siheng Li` and 15-668

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT:
OpenMP - Shane
OpenMPI - David
SIMD - Shane
Discussion of methods - Both
Data collection - Both
Speedup analysis - Both
Report and poster - Both
Distribution - 50%/50%