15-418 Final Project Report

Larry Geng (lgeng), Aleksander Tarczynski (atarczyn)

Summary

We implemented and compared 3 different parallel versions of a 2D-game dungeon generation algorithm based on a sequential outline found online. Parallel versions were run on the GPU with CUDA, and on the CPU with ISPC and OpenMP. Our results showed that a GPU-parallelized version of the algorithm can run 600x faster than our sequential version, with trade-offs in poor placement of dungeon main rooms. CPU-parallelized versions provided substantial runtime improvement over sequential with less egregious main room pushing.

Background

Many 2-dimensional video games use “dungeons” as their setting. Players move through rooms in the dungeon as they play through the game. Each room is often a different size or shape, with rooms being connected by hallways and other small rooms in-between.

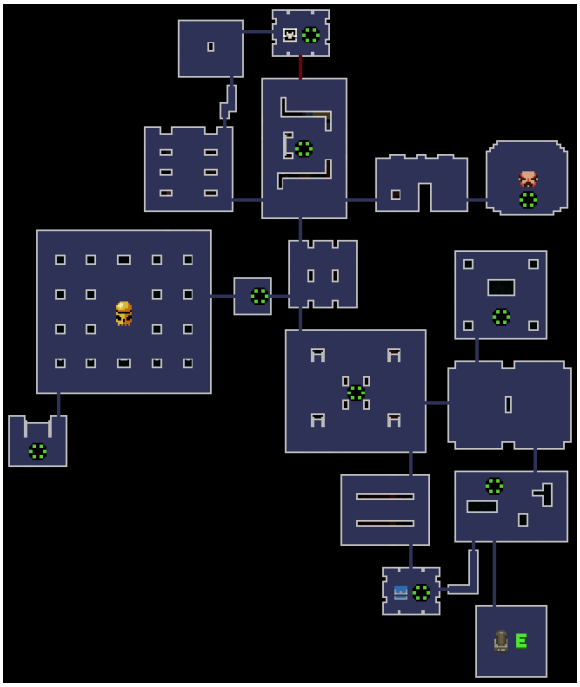


Figure 1: Example dungeon map from “Enter the Gungeon”(1)

Sequential Algorithm Description

Some games use algorithms to generate dungeons procedurally. One algorithm we found for generating dungeons had no dependencies on libraries, and fit the scope of the 418 final project (2). It is this algorithm that we want to accelerate by developing parallel solutions. The input to the algorithm is the number of rooms we want to use as a baseline, and the output is a struct containing an array of rooms and an array of hallways.

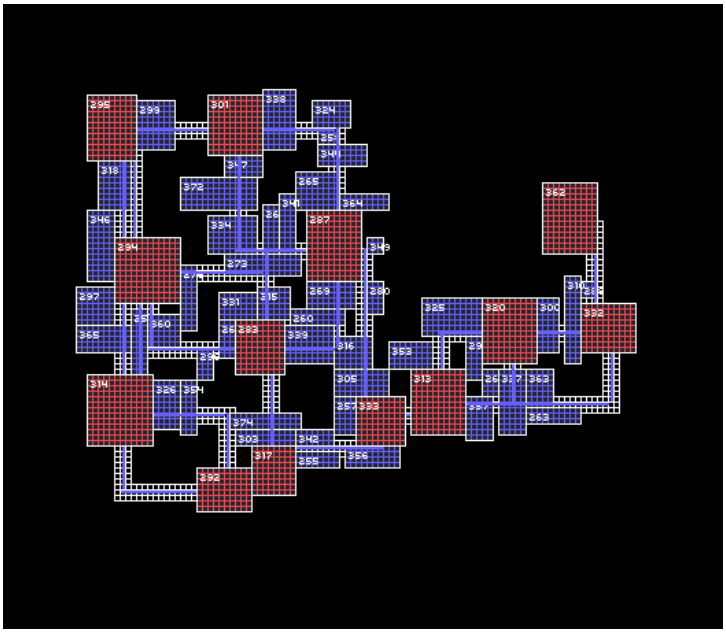


Figure 2: Example generated map using our chosen algorithm.(2)

The algorithm involves many sequential steps. At a high level, these steps are:

1. Given an input number of rooms, generate randomly sized rectangles placed in a circle
2. Push the rectangles away from each other until they no longer overlap (separation)
3. Filter out the “main rooms” of the dungeon
4. Construct a graph of the main room center points using Delaunay Triangulation
5. Generate a minimum-spanning tree for the graph
6. Add back some edges from the original triangulated graph
7. Map out hallways connecting the main rooms
8. Fill in the hallways using both leftover small rooms and additional tiles, as needed.

By parallelising this dungeon generation algorithm, we hoped to enable more flexible usage of procedural dungeons. For example, a fast dungeon generation algorithm could let games create

enormously large dungeons. Games could also generate many smaller dungeons at the same time. Having such options could help game creators better express their ideas.

When writing the proposal for this project, we were not sure whether doing the generation on the GPU is worth the expense of copying data. One of our goals for this project was to evaluate the performance difference between CPU and GPU for dungeon generation.

Separation Steering

After designing a functional sequential version of the algorithm, we noticed that the room separation step can take 100 to 1000 times as long as the other steps. Therefore, we decided to focus all of our parallelization efforts on the separation step. Even in our fastest running parallel implementations, separation easily made up a majority of the runtime, so we felt that we were justified in focusing on it.

Another aspect that made the separation step interesting is that it was not a commonly known algorithm. In our early research during the project, we found parallel implementations for minimum spanning tree and Delaunay triangulation, which were the other two steps we expected to take a long time. When looking online for separation steering examples, we found only sequential algorithms.

The main data structure involved for separation steering is a list of rectangles. Each rectangle, corresponding to a candidate room, contains its x and y center coordinates and its height and width.

The gist of the algorithm is that, while any two rectangles are overlapping, the program should loop through all pairs of rectangles. For every rectangle pair that overlaps, the two rectangles should be moved away from each other.

Separating the rooms using this method means that each room is essentially dependent on every other room. There is also, however, a lot of potential parallelism over each room pair. While in concept there is spatial locality in that overlapping rectangles are next to each other, we found it difficult to exploit this spatial locality since the rectangles can move quite a bit over the course of the program. The algorithm is somewhat amenable to data-parallelism and SIMD, but a true sequential algorithm may produce the most compact separation due to every room movement step being able to build on every prior room movement. Transforming the algorithm into something that is data-parallel could result in each room movement being reliant on outdated or irrelevant data.

Approach

In implementing the dungeon generation, we used code for Delaunay triangulation (3). While we initially intended to write both sequential and parallel triangulation code on our own, we shifted our focus to the separation steering.

ISPC

A direct approach of parallelizing the nested loops proved to be sufficient for the ISPC implementation. Due to the inner loop code writing to the room struct, it was necessary to offset each thread in the gang such that no threads would write back to the same room. The biggest issue with said problem was to find a separation style that would have each gang of threads run without iterating over the same room twice. We found a solution by having each thread in a gang take one a separate iterator of the outer loop. This was still not a perfect solution, but we concluded it would still achieve a working result every time.

The revelation that we had was with the correctness of each iteration, specifically dealing with race conditions. A race condition could occur if one room moved to a previously not overlapping position that would now be overlapping. This issue would solve itself on the next iteration of the separation algorithm, since the two rooms that were intersecting could be moved apart again. We made use of this idea to avoid synchronization for both ISPC and every other parallel implementation.

We found that the ISPC version had a significant amount of divergent execution, especially towards the later stages of the algorithm where the likelihood of only 1 or 2 of the 8 threads finding a pair of overlapping rooms drops dramatically.

We also used ISPC tasks to get the full performance. Due to the 1-dimensional nature of the data structure we were iterating over, it was simple to set up a tasking block to separate the work. We found that a number of tasks that evenly divides the workload, at around 200 total tasks, had the best performance.

CUDA

For our CUDA implementation, we wanted to be able to run the dungeon generation on a GHC machine and make use of the RTX 2080.

The most obvious way to leverage a GPU is to map each CUDA thread to a room. Then, each thread can check all the other rooms to evaluate both overlap and room movement. One immediate difficulty is that for any overlapping pair, the algorithm requires both rooms to move. While we attempted to avoid any conflicting threads in the ISPC implementation, we tried to brute force everything in CUDA.

Our initial attempts to build a CUDA implementation failed to scale beyond 256 threads. Since we were using 256 threads per block, we believed that the performance suffered due to the extreme amount of sharing between blocks.

To alleviate this problem, we separated the overlap kernel from the data movement kernel. Although each thread would individually calculate pairwise overlaps for determining whether or not to move rooms, we used a basic parallel reduction for calculating the loop guard of the outer while-loop. Our implementation only used a true parallel reduction within each thread block and updated a global overlap flag using atomic ORs. We felt that this balanced speed with complexity, since we were unlikely to use an extreme amount of blocks.

We also attempted to exploit spatial locality by having each block separate a group of 256 rooms. Then, we separated each group of 256 rooms, treating each group as a large rectangle. While extremely fast, this method produced rather poor results. The smaller groupings of rectangles were extremely obvious and separated. Since the more naive brute force method worked so well, we did not pursue this idea further.

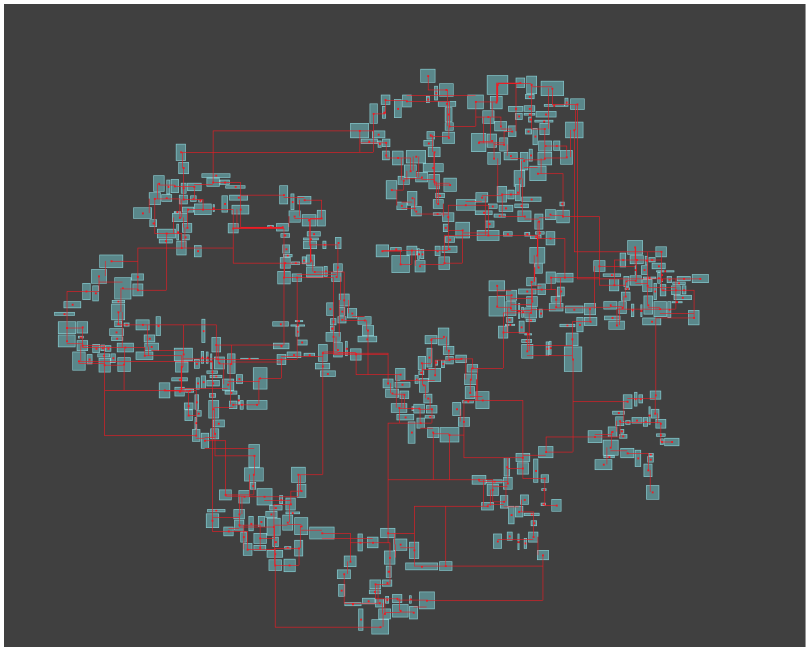


Figure 3: 1000 rooms, Grouped room CUDA implementation, main and hallway rooms, group size of 64

OpenMP

While working on the other two implementations, we felt that OpenMP would be a very natural fit for the separation algorithm. Given the extreme amount of data sharing going on, the shared memory abstraction seemed like a very simple way to parallelize over the room overlap and room movement. We only had to add a single “pragma omp parallel for” to our code, and saw very good near-linear speedup at high room counts.

Results

Separation Steering Percentage

In the table below, we have included some data on the percentage of runtime that is spent on separation steering. As we stated earlier, the extremely high percentages are why we focused solely on parallelizing the separation algorithm.

Performance Measurement Method

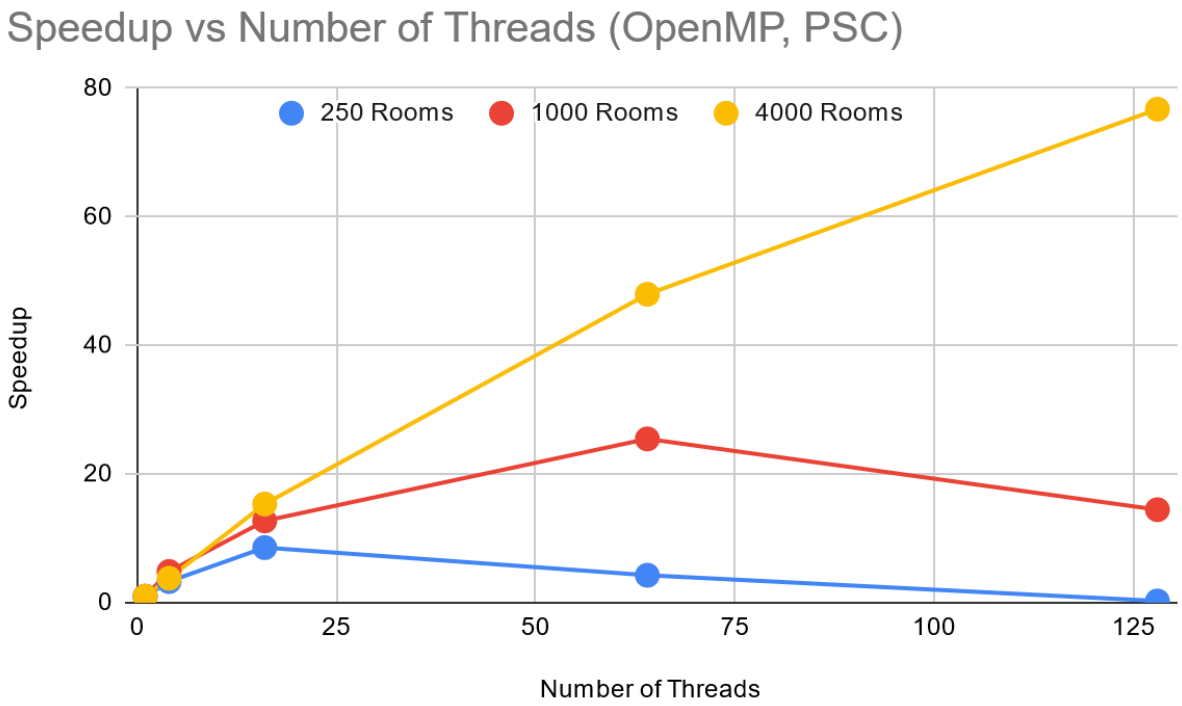
To measure performance, we primarily used speedup in comparison to sequential execution on the PSC machines using identical input. The sequential execution was equivalent to the OpenMP implementation run on one thread. We ran all of the measurements for the ISPC and CUDA implementations on GHC 80, and we ran the measurements for the OpenMP implementation on a PSC RM machine. For the OpenMP implementation, we gathered data using 1, 4, 16, 64, and 128 threads.

We gathered data using 250 rooms, 1000 rooms, 4,000, and 10,000 rooms as input. For the 10,000 rooms input, we only gathered data at 64 and 128 threads in the OpenMP implementation. As a result, we present wall clock time rather than speedup when comparing performance on 10,000 rooms.

While gathering performance data, we noticed a lot of variability from run-to-run. Since all of our parallel implementations were non-deterministic, we saw runtime vary as much as 10-15% between runs. The following results are single data points for each configuration, so we believe that some of their stranger aspects (such as superlinear speedup for 4 OpenMP threads) can be explained by the runtime variance.

Speedup Results

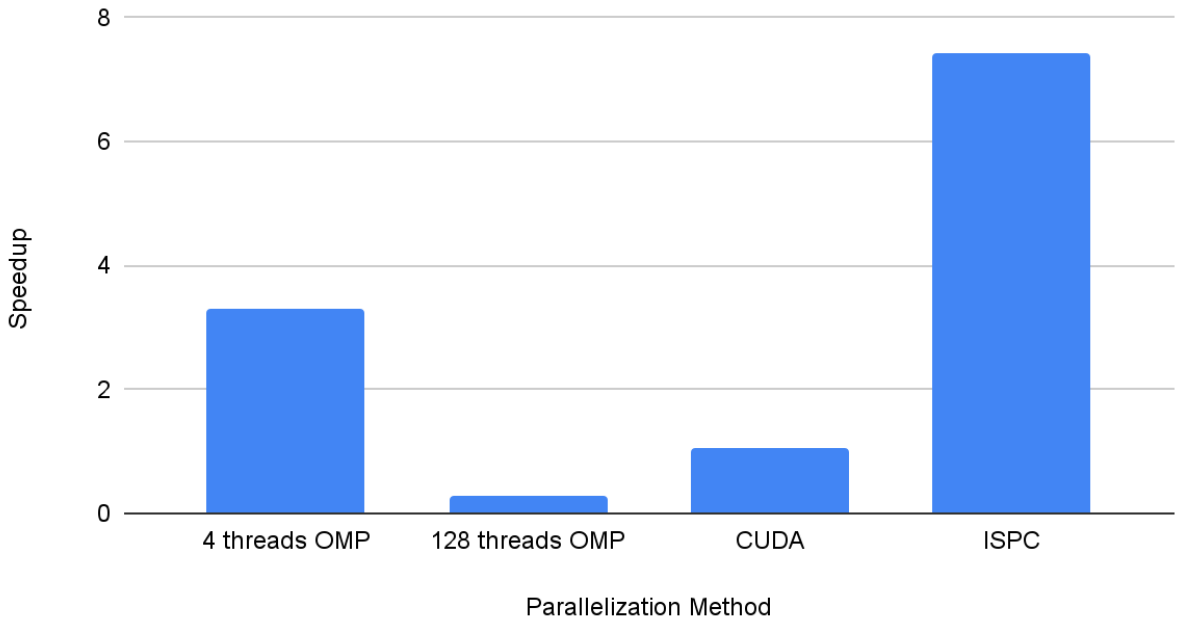
We saw very good speedup for the OpenMP implementation when the number of rooms was high.



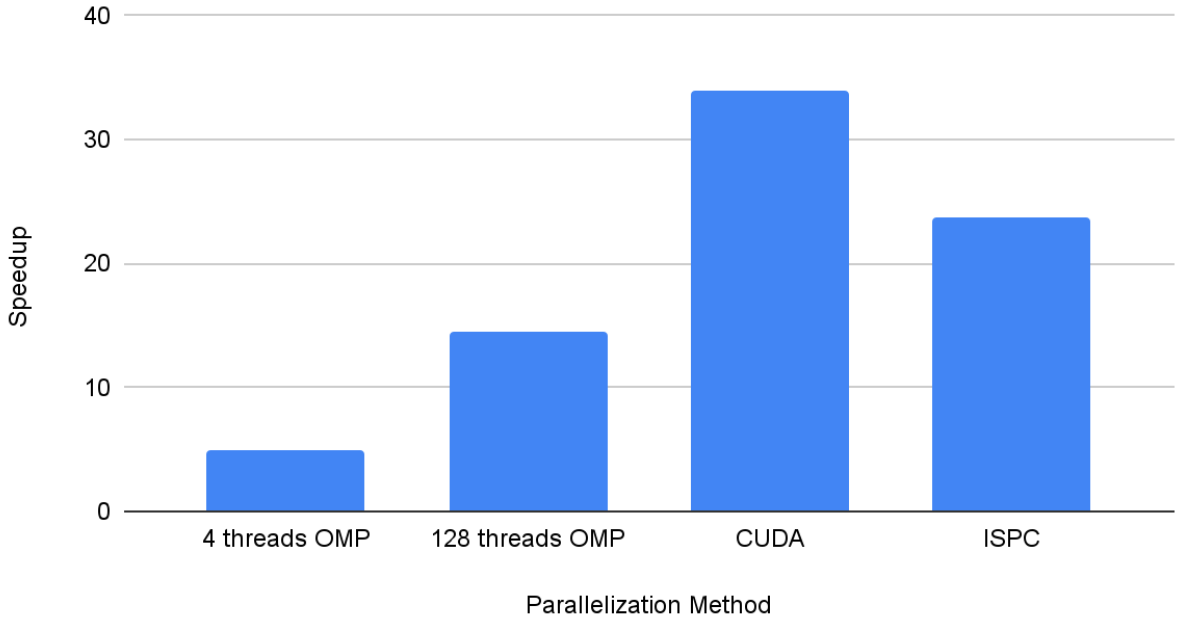
At 250 rooms and 1000 rooms, the speedup degraded after 16 and 64 threads respectively. We believe that this degradation was caused by the increased amount of communication that came with lower room counts. With fewer rooms, threads are more likely to begin working on the same room. The speedup at 4,000 rooms is nearly linear, which we found very impressive.

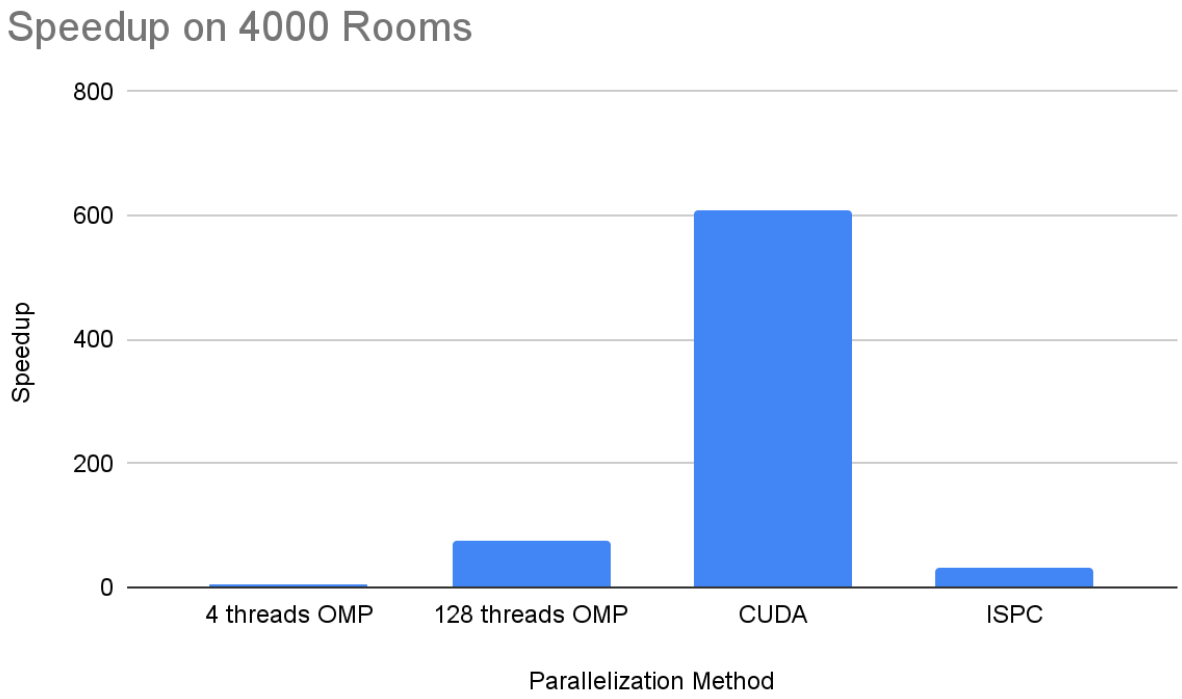
In the following graphs, we use one bar chart for each room size to accommodate the wide range of speedup numbers.

Speedup on 250 Rooms



Speedup on 1000 Rooms

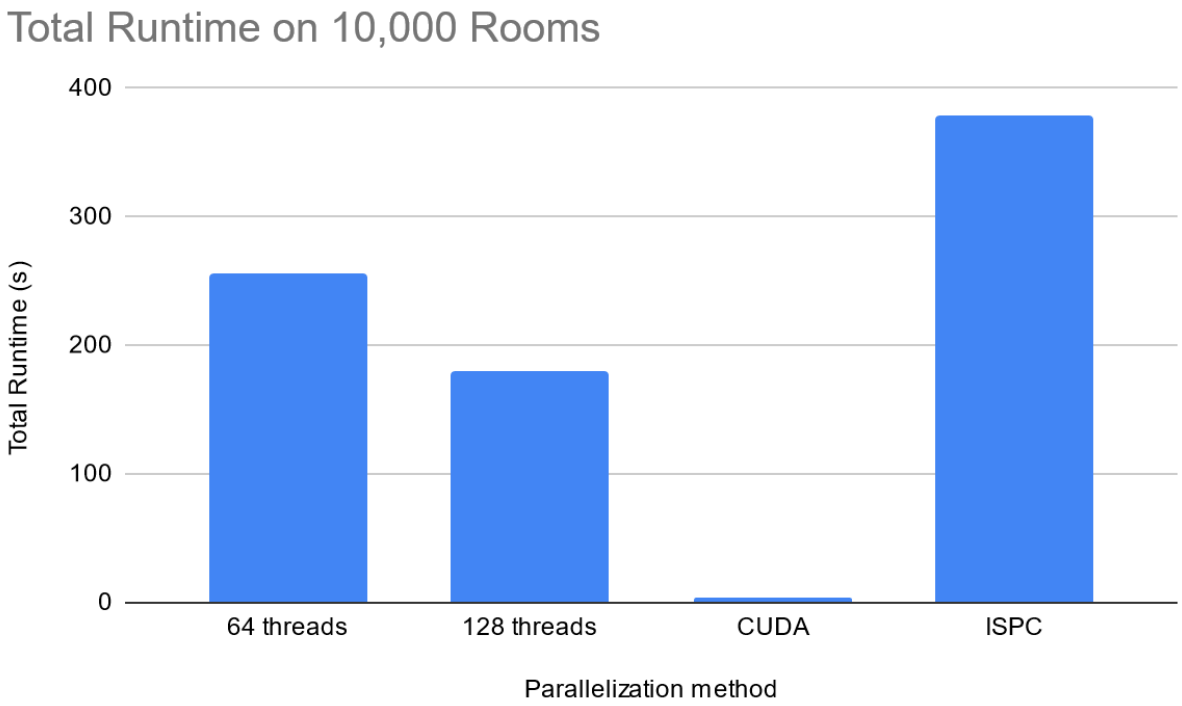




We were satisfied with the speedup results that we got from CUDA and ISPC. On 250 rooms, the overhead in copying data to and from the GPU prevents CUDA from seeing strong speedup. ISPC, while also using SIMD parallelization, does not have to copy data and therefore produces the best speedup.

This pattern reverses somewhat with 1000 rooms, as CUDA overtakes ISPC in speedup. By 4000 rooms, CUDA produces speedup far ahead of any other implementation. These numbers exceeded our expectations. Based on our implementation, we expected to see substantial divergent execution in both SIMD implementations. The extremely parallel nature of the problem, as well as the amortization of the data movement cost, means that CUDA ended up achieving up to 600x speedup over sequential execution.

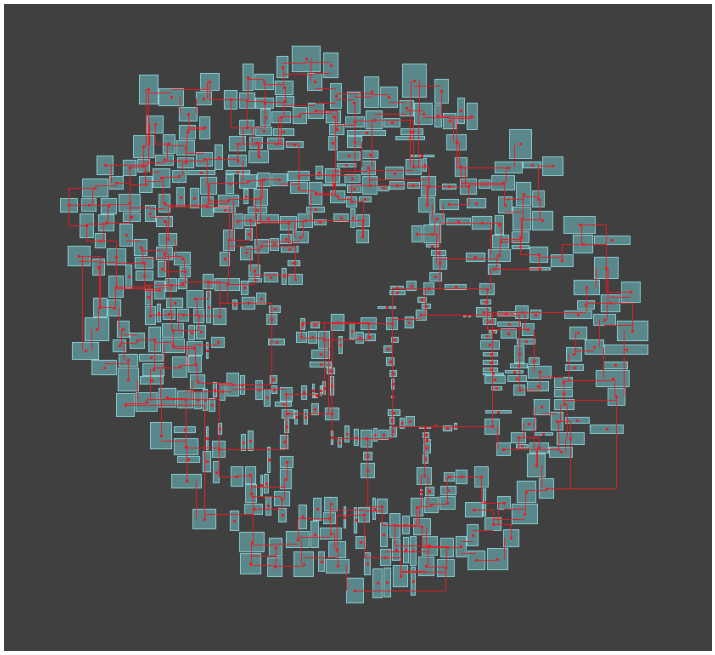
OpenMP overtakes ISPC in speedup at 4000 rooms. Since we tested ISPC only with the GHC machines, we believe that it was limited in core count. When compared to OpenMP with 4 threads, ISPC still does substantially better. Assuming that the ISPC does 8-way SPMD with one functional unit per core, the best-case speedup we saw of 30 is relatively close to the ideal of 8 x 8 = 64.



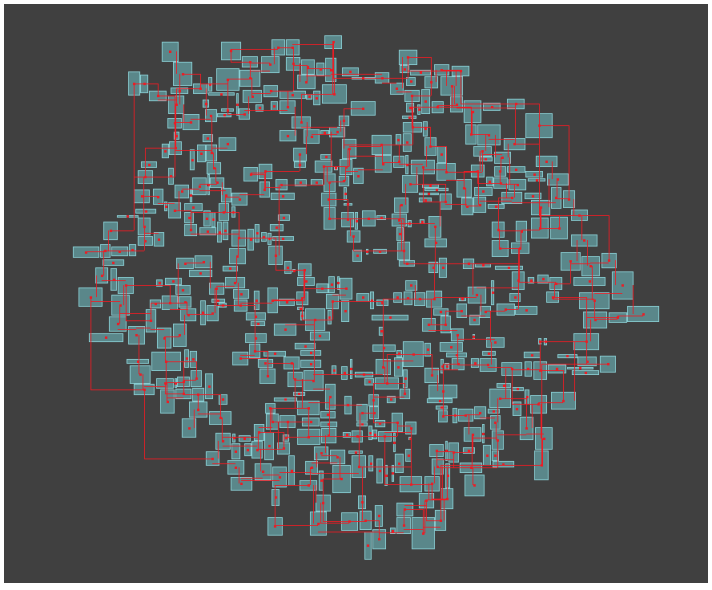
At 10,000 rooms, we see approximately the same pattern as at 4000 rooms. Note that the above graph displays runtime. CUDA performs by far the best, while OpenMP with 64 and 128 threads slightly leads ISPC.

Separation Quality

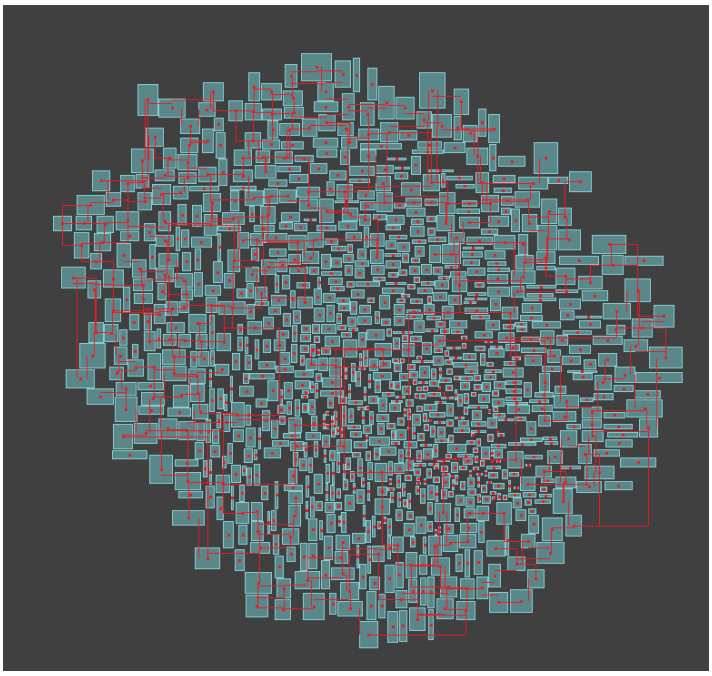
While examining the results from separation, we noticed that the CUDA implementation produced results with very large gaps between main rooms. All of the larger rooms were on the outskirts, while smaller rooms that ultimately would not be added as a hallway or main room filled the middle. Although other implementations saw similar patterns, the effect was much more pronounced in the CUDA implementation.



1000 rooms, CUDA implementation, main and hallway rooms shown



1000 rooms, OpenMP with 8 threads, main and hallway rooms shown



1000 rooms, CUDA implementation, all rooms shown

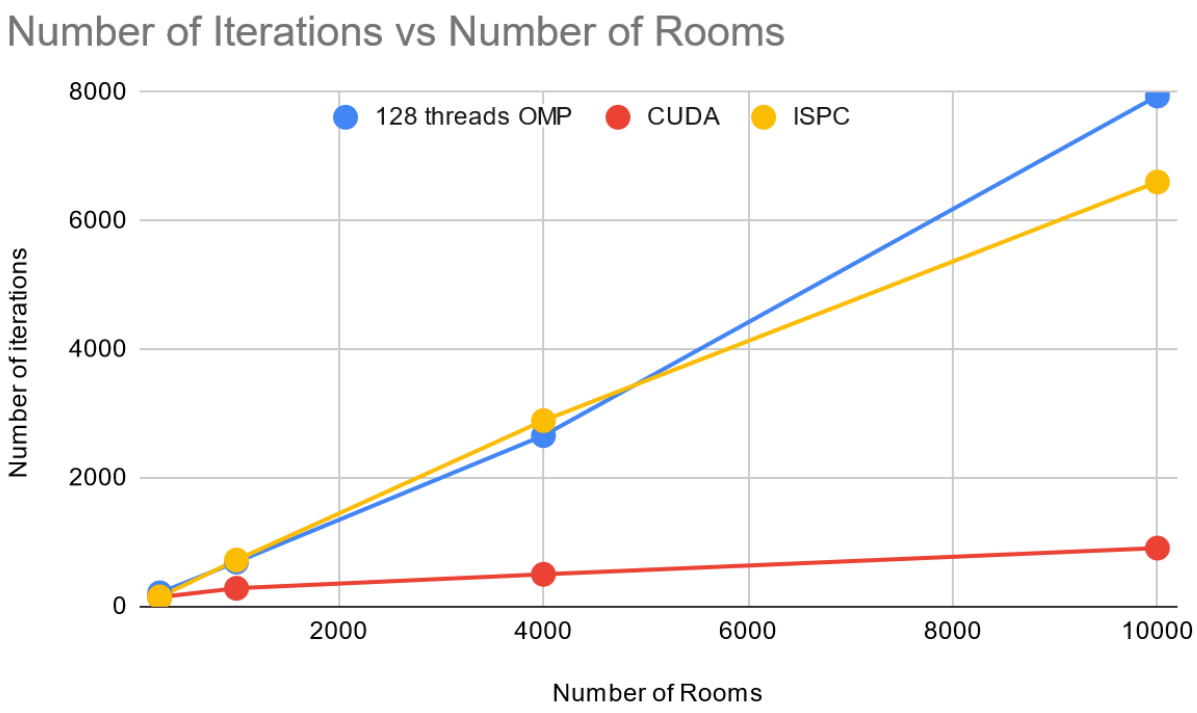
At first, we thought that the CUDA implementation could be separating rooms further from each other. We attempted to create a “separation quality” metric that would roughly quantify how far apart the resulting rooms are. The metric simply found the area of the rectangle that bounded all of the resulting rooms. Our testing showed that while the separation quality varied a fair amount from run-to-run, we did not see any distinction between the CUDA and the other implementations.

	4 Threads OMP	**128 threads OMP**	**CUDA**	**ISPC**
Separation Quality (1000 rooms)	518040	481988	469426.25	508725

One major difference we saw between CUDA and the other implementations was that CUDA ran far fewer iterations of the outer while-loop in the separation algorithm. Although the iterations are not directly equal between implementations, one iteration should check for overlap between every pair of rooms twice, regardless of the implementation. When there are no more overlaps, the separation algorithm ends.

We speculate that the extremely parallel nature of CUDA execution means that more threads are using outdated data at any one time. The result could be that larger rooms, which are more likely to overlap, are pushed further than they would be in other implementations. By pushing larger rooms faster, the CUDA implementation takes fewer iterations to remove all overlap.

Additionally, this mechanism would explain why the CUDA implementation pushes noticeably more large rooms to the outer edge of the dungeon.



Conclusion

For generating dungeons with 250 rooms, ISPC showcased the best speedup. By using the vector execution units on the CPU, the ISPC implementation did not have to incur the cost of copying data to and from the GPU. On the other hand, the CUDA implementation shined at higher room counts, producing impressive speedup numbers when it was able to amortize the overhead of using the GPU.

OpenMP parallelization performed solidly overall. When using only 4 threads, OpenMP produced almost linear speedups at the three different room counts we tested. Although 16 or more threads degraded performance at 250 rooms, they performed well at higher room counts and we again saw near-linear scaling.

Dungeon generation, however, is an algorithm specifically for games. We doubt that most gaming computers will have more than 16 threads to utilize. Therefore, we think that either ISPC or CUDA is the best way to go, depending on how many rooms are being generated in the dungeon.

Throughout the project, we learned the importance of visual aids when debugging and analyzing code, especially in situations where there is no real “correct” result for an algorithm. In our case,

developing an animation to show each separation step gave insight into the differences of the three parallelization methods. It also proved as a fun way to show our work off to those who might not understand the inner details of our project.

Overall, we are quite satisfied with our project. We tried three different parallelization methods and saw unique benefits from each one.

References

1. Dungeon Generation in *Enter the Gungeon*, https://www.boristhebrave.com/2019/07/28/dungeon-generation-in-enter-the-gungeon/
2. Procedural dungeon generation algorithm used, https://www.gamedeveloper.com/programming/procedural-dungeon-generation-algorithm .
3. Clarkson-Delaunay triangulation code, https://www.codeproject.com/Articles/587629/A-Delaunay-triangulation-function-in-C

List of Work

Algorithm / Project Research: Aleksander 50%, Larry 50%; 3 hours total
Sequential Algorithm: Larry 70%, Aleksander 30%; 10 hours total
UI / Animation Development: Aleksander 100%; 20 hours total
OpenMP: Larry 100%; 2 hours total
ISPC: Aleksander 100%; 15 hours total
CUDA: Larry 100%, 12 hours total
Video Presentation: Aleksander 50%, Larry 50%; 1 hour total
Data Collection and Final Report: Larry 70%, Aleksander 30%; 8 hours total
Total Hours: 71 hours