Parallelization of Procedural Content Generation

Rachel, Hunter, Cay, Ben, Ryan

I. Abstract

The findings of the combination of various algorithms with multithreading implementations was an overall increase in the ability to generate large quantities of complex data. Perlin and Worley noise was able to reduce time in a multithreading situation for large images although if the noise to be generated is a small size then the linear implementation could be considered to be an option to reduce the complexity introduced by the threads. Marching cubes was also a multithreading success, while wave function collapse did not prove to have significant advantages to using multithreading in implementation. The size and scope of the image generation proved to be a critical factor to consider when determining whether to use a multithreaded implementation of an algorithm or not.

II. Introduction

Our team is interested in taking parallelization into the game design field. Looking to games like Minecraft for inspiration we aim to investigate terrain and mesh generation broadly known as procedural content generation. Though we are programmers with different skill sets, we hope to reach a greater understanding of procedural content generation by sharing our knowledge and enhancing our individual work to eventually produce a unified conclusion. We will conduct our research through parallelized algorithms commonly used for such content generation.

III. Problem Statement

When creating video games, simulators, and other 3D world environments, a common problem arises with procedural content generation (PCG). Large landscapes, complex textures, and large-level assortments can be a taxing computation load on a computer and lead to difficulties that extend the time needed for game development. Hastening these procedures is a priority as software that utilizes content generation largely suffers from even minute delays. This can cause issues with simulation accuracy, hinder the player experience, and cause frustration on the side of both the developer and the player.

IV. Challenges

Finding good metrics to cross-compare these algorithms and combinations of certain algorithms has been and will continue to be a challenge. Some like Perlin noise and Worley noise are better suited for generating terrain while the marching cubes algorithm is to make meshes. These algorithms suit various different purposes, and the efficiency of our written programs may not agree with the full potential they have to offer, causing the data we retrieve and compare to be subjective. Overall our goal is to see how these improve content generation as a whole so we will include these differences in our research.

We also had a few challenges with our programs deciding how to implement multithreading and in some cases, it took a long time for the programs to execute. This wait time was only sometimes alleviated with multithreading. It was also difficult to navigate the four complex algorithms and some certainly got more attention than others and were more effective. Another challenge was implementing our code together and making sure it ran seamlessly.

An additional challenge was running out of heap space for large resolutions in the noise generation and marching cubes algorithms. Larger resolutions required finding a computer with enough RAM to perform experimentation.

V. Tasks

Our tasks were to continue to refine our code to collect comparison points for the research. Each member wrote one of our implemented algorithms and will work to see how parallelization improves runtime, the complexity of generation, etc. We have these producing output images respectively for further data points of comparison. We also had a member focus on project manager roles and keeping us on track, as well as someone taking on a lot of the writing in order to communicate our findings effectively. Dividing up the workload really helped us implement a variety of complex algorithms in the time allotted while balancing other workloads.

VI. Goals

Every week we set ourselves a goal. We each aimed to write one page or so on the findings we specifically discover with our respective code. We also had a goal to conclude the benefits and drawbacks of each content parallelization algorithm that we are working with. Overall, our project goals were met with good progress and overall led to successful research and findings as well as a viable product in the GitHub repository.

VII. Implementation

Our roles as follows were not limited by considering mainly of the following assignments although combining the code was a team effort and 3d images and gifs were thanks to the genius of Cay and Hunter primarily.

Rachel - Perlin Noise & Writer

Hunter- Worley Noise

Cay - Marching Cubes

Ben - Researching Parallelization and comparability, Project Manager

Ryan - Collapsable Wave Functions

Each member of our team worked to implement an algorithm under the guidance and help of Ben, our project manager. We implemented these algorithms without parallelization first so that a comparison can later be made.

One challenge is that we produce output images based on the data the algorithms create. For Perlin noise, this could simply be converted to a pixelated grayscale image. For Worley noise, each sheet PNG was stitched together to create a stream of images or a nice video.

Since the marching cubes algorithm generates a mesh, OpenGL was used in order to have a visualization. Cay had some difficulty with this but ultimately overcame it and was able to get it working.

For Wave Function Collapse, this involved researching existing implementations, modifying them, and parallelizing them. The output from this algorithm is a 2D image.

It's worth noting some members were not very familiar with Java so deciding to use it meant they had to learn a little and get the environment set up. This was a bit confusing as Java packages always seem to cause a kerfuffle but we managed to all get set up.

VIII. Technique

Each algorithm was implemented with a unique technique per the person implementing in order to best simulate a different facet of parallel content generation with multithreading for terrains and meshes. The following details the specifics of each of the algorithms implementations and findings.

A. Perlin Noise

Parallelizing Perlin noise generation really helps for large datasets. The generation is a computationally intensive task, as it involves the calculation of noise values at each point in the grid. The algorithm takes as input a certain number of floating point parameters (depending on the dimension) and returns a value in a certain range [1]. Parallelization can be achieved by dividing the grid into smaller chunks and assigning each chunk to a separate processing unit, such as a thread or a computing core. Each unit can then independently calculate the noise values for its assigned region, allowing for concurrent execution. This parallel approach can significantly reduce the overall computation time, especially on multi-core processors, by taking advantage of the available parallel processing power.

Additionally, parallelization of Perlin noise for three-dimensional or higher-dimensional spaces, where the computational load increases substantially, is really useful. By distributing the workload among multiple processors, the overall time required to generate the noise for the entire space is reduced. However, it's important to note that parallelization introduces challenges such as managing multiple threads and accessing data, which need to be carefully addressed to ensure correct and efficient execution. Clearly, this optimization will help overall with the generation of grid content such as terrain of mass size in a quicker time.

Since multithreading is complex and intensive it was found that for small images of Perlin noise, it may be faster not to use multithreading since it adds complexity and hurtles. But if generating multiple images multithreading will drastically help on generation time.

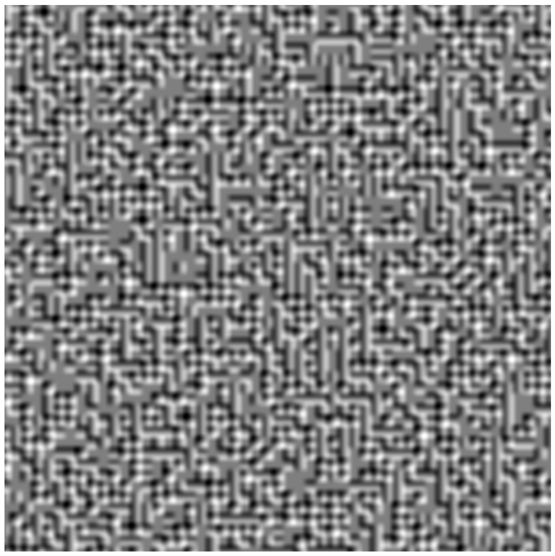

Fig. 1. Perlin Noise

B. Worley Noise

Parallelizing Worley noise generation, also known as cellular noise, is pretty similar to Perlin noise. In Worley noise, each point in the grid corresponds to a cell, and the computation involves determining the distance to the nearest and second-nearest neighboring cells for every grid point. This process can be parallelized effectively by dividing the grid into segments and assigning each segment to a parallel processing unit. Each unit can independently calculate the Worley noise values for its assigned region, resulting in concurrent computation.

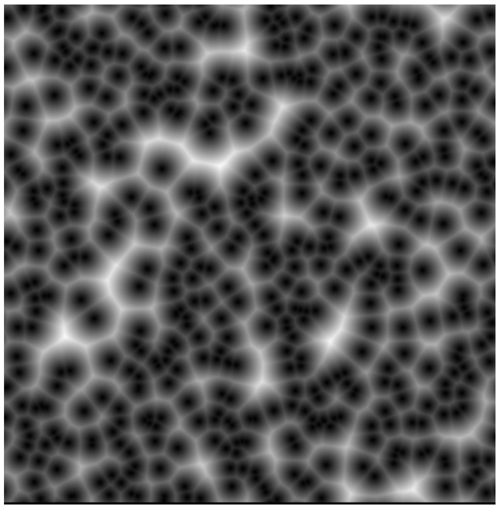

Fig. 2. 4096^2 Worley Noise image with 500 points. Generated using multithreading in 4.25 seconds.

Parallelization becomes even more critical in scenarios where real-time or interactive applications, such as procedural terrain generation or texture synthesis, require Worley noise. By leveraging parallel processing capabilities, such as multi-core CPUs or GPU shaders, the generation of Worley noise can be significantly accelerated, contributing to smoother and more responsive user experiences. Synchronization and load balancing are key here to ensure the execution of Worley noise generation algorithms across multiple processing units.

For understanding implementation, the scatter points across the space to create a distribution. For each cell, calculating the distance from all points will need to occur. Then each cell will be assigned a value corresponding to the distance from its nearest point. It then must normalize all values by dividing each cell's value by the largest cell value in order to produce the images of worley noise.

However, the computational time increases significantly as the number of points or dimensions increases. For example, generating a 3D 1000x3 texture with 500 points took 36 minutes and 41.616 seconds in testing. This slowdown occurs because each cell, out of the billion cells in a 1000x3 texture, is calculated against each of the 500 random points, resulting in an immense number of calculations. So there are definite limitations to what can reasonably be done with this method and the number of cores and memory we had access to for this research.

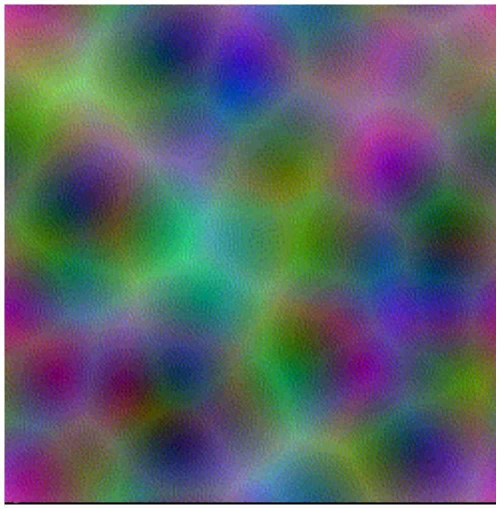

Fig. 3. 3D Worley Noise with texture wrapping and seamless tiling, also as a GIF format

C. Marching Cubes

Marching Cubes is an algorithm for rendering volumetric data as an isomorphic surface, or triangle mesh. It's commonly used in medical visualizations such as rendering CT or MRI scans or other types of 3D models. In our case, we can use it to visualize Perlin noise or Worley noise data as isomorphic surface. It works by "marching" through the cubes in a voxel grid, one by one. Each cube has eight points whose values are generated using one of the aforementioned noise generators or a scalar field, one for each corner. Depending on the values of these corners, it will create one of 256 different configurations (2^8 = 256) by treating each of the 8 values as a bit in an 8-bit integer. Since a bit can have one of two values, that create 2^8 possible configurations. Whether a bit is flipped on or off is determined by which points on the voxel are "below surface level" or active. From this, it consults a pre-calculated triangulation table of the 256 configurations using the active corners to get the vertex positions for the mesh's triangles.

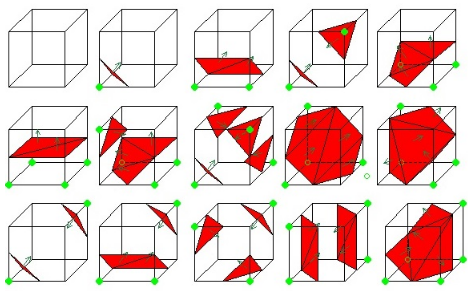

Fig. 4. Marching Cubes Triangulation Table [2]

For a grid size of 1024^3, this would require 1024^3 number of calls to the marching cubes function x 15 vertex position calculations per (x, y, z) coordinate. This is 24.6 billion computations in total. Increasing the grid size would increase the overall complexity cubically.

We chose to render these triangles using OpenGL. This algorithm has a runtime of O(n^3), and thus can take very long at high resolutions (> 128 voxels wide). Parallelization is done partly by the GPU when rendering the mesh; however, parallelization could potentially improve runtimes at higher resolutions. How this may be done is by either chunkifying the grid into 9ths, 27ths, etc. and assigning a thread to each or using shared variables (locks for mutual exclusion and atomic variables for the cube coordinates).

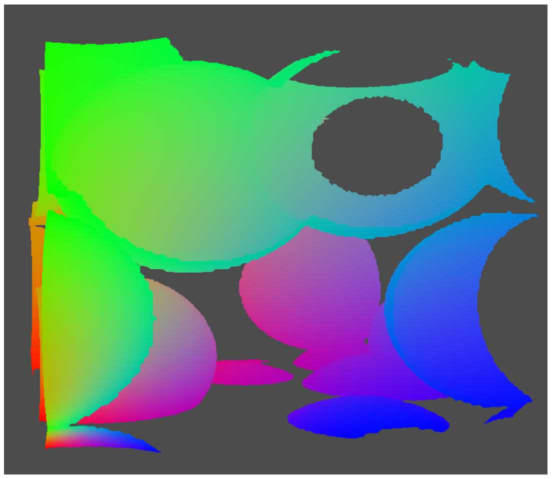

Fig. 5. Rendering a 256^3 grid of Worley Noise data using Marching Cubes

The shared counter approach first locks the coordinates to the next cube that needs "marching". In Java, this uses an AtomicInteger for enabling shared read-write operations. It then calculates the mesh configuration for a cube outside of the critical section. After doing so, it enters another critical section to add the positions to the buffer to be sent to the GPU. In Java, I used a CopyOnWriteArrayList<Vector3f> to store the positional data that could be mutated by multiple threads.

The chunkification approach takes a voxel grid and splits it into chunks. Each thread takes on a chunk and iterates over it inside a triply nested for loop. In Java, I used a hash map to keep track of which thread was in charge of which chunk for merging the positions later in the correct order to be sent to the GPU.

D. Wave Collapse

The Wave Function Collapse is an algorithm for generating images based on existing data, using rules deciphered from input images as the basis to create a new image that follows those rules. The first part is the creation of rules from the original image, with the algorithm going through every pixel and determining the relation to surrounding pixels. In simple models, only the directly connected pixels are taken into account, while in overlapping models rules take into account pixels across the whole image for determining what can be connected to a specific pixel. The implementation in Java was modified from an Open Source Repository from Allison Casey [3]. This itself is based on a C# base, also Open Source, that was also used for this project from Maxim Gumin [4].

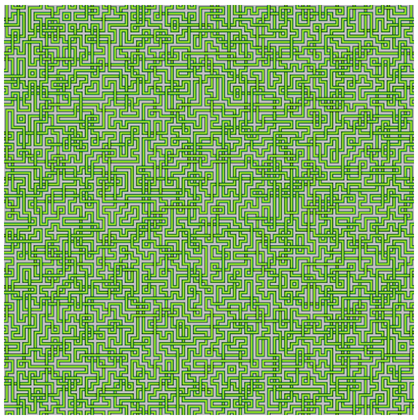

Fig. 6. Dense Generated Image

Then, the algorithm creates a blank canvas of a size determined by the user and starts at the top left. The "wave function" exists at this point in a state where that pixel can be any of the colors from the original image, and it will then "collapse" down to one specific color in that pixel. This is typically chosen at random to begin with, among the colors that, by the rules, can be at the edge of the canvas. Then, the wave function algorithm propagates outward to pixels connected to that one, with the wave function collapsing to pixels that are able to be connected to it. This continues until either the canvas is filled, in which case the algorithm succeeds, or it reaches a point where it cannot possibly find a color to use that works within the established rules. At this point the algorithm restarts from that very first pixel, collapsing the wave function down to a different choice and propagating from that new starting point. The wave function algorithm would fail if no image of the size and parameters specified could be created from the established ruleset.

The Parallelization approach used for the WFC involves multithreading being applied to the propagation algorithm. A lock-free queue is used to keep track of changed nodes/pixels instead of the original array used before parallelization. This implementation was inspired by a C++ project from 2019 that had found success in this approach [5]. The outer loop of the propagation algorithm (Loops through all changed nodes across the process) assigns each node to a new thread that runs the inner algorithm sequentially. A deeper approach, applying multithreading to every layer of loops of the algorithm, was considered but presented mixed results. This approach also only works well with a Simple Tiled Model, as in Overlapping Models a fail state is encountered more frequently if multiple nodes are collapsing at once, leading to the program getting caught in an endless loop.

Parallelization replaces the original program's array for storing changed nodes with a lock-free queue and assigns every changed node to a new thread to handle the propagation algorithm. The algorithm could be layered with more multi-threading, but the results were mixed. Parallelization in other algorithms, such as the observation algorithm, faced similarly mixed results.

Overall, the results were not what was expected. Multi-threading generally did not speed up the program, and in fact, potentially due to redundant uses of the same pixels multiple times, the algorithm ran considerably slower and crashed more frequently. The sequential WFC algorithm is simple and efficient enough to where it seems that parallelization doesn't provide much benefits, and on smaller images like the ones used in experiments, will instead clutter the program and slow it down.

IX. Evaluation

Evaluation is based on data size of generation, time of generation, and overall complexity of the generated content. Also of note is the difference between the original, unparalleled programs, and their more efficient counterparts. Each algorithm will be weighed on its own merit and ultimately will be compared against each other in regard to the earlier mentioned criteria.

Perlin and Worley's noise proved to be effectively generated with reduced time using multithreading, however, for Perlin noise was not in all cases. If the content being generated was smaller, it actually took longer to multithread as it was not a straightforward approach. Generally, a large amount of content will need to be generated however and for these cases, multithreading is a game changer.

For the Wave Function Collapse algorithm, parallelization generally slowed the program down, though it can be assumed that a large enough image may have benefited from it should the sequential algorithm take too long. In all experiments, however, the multi-threaded algorithm, even with the addition of a lock-free queue, took considerably longer than the sequential model, which usually ran in well under 10 seconds for anywhere from 72x72 to 128x128 pixel images. The benefits of parallelization so far seem to have been outweighed by the time cost of creating and managing threads and managing the queue. Overlapping Models also hang indefinitely using this approach, due to the use of an entire image in the propagation step.

For the Marching Cubes algorithm, parallelization had a noticeable benefit. The first approach of using a shared counter method proved to worsen run-times. The calculations done outside of critical sections were not complex or lengthy enough for there to be any benefit towards having multiple threads work on the same voxel grid. However, the chunkification approach, where each thread had their own voxel grid to work with, greatly reduced run-times, increasing speeds by 64.6%. This is 2.81x faster than the single-threaded implementation. However, larger resolutions required large amounts of memory.

X. Discussion

Before this project, several members of the group had taken CAP 4720 Computer Graphics and had taken a passing interest in matters directly or tangentially related to procedural content generation. Parallelization seems particularly suited to aid in this process. As research progresses, it will be incredibly interesting to see just how much content generation can be made more efficient and which algorithms tend to benefit more from its aid.

Following the research we considered the threshold for when content moves from being fastest parallel to faster multithreading and how this threshold continues for adding more cores and complexity moving forward. The size and scope of the content being generated is the deciding factor.

Generally speaking the simpler the sequential model, the less benefits offered by parallelization. An efficient sequential model can be muddied by multi-threading, but a more complex sequential model could be massively improved. Parallelization showed considerable benefits to more complex, processor-and-memory-intensive algorithms, while showing drawbacks to simpler algorithms.

XI. Conclusion

While multithreading can improve some algorithms for PCG it wasn't the most effective for all situations. Ultimately it depends on the the size and amount of content as well as what algorithms are implemented in order to determine if multithreading will be the best choice for the implementation of the process. It was very effective for generating large quantities of Perlin and Worley noise, however, we did not find it to be as effective for wave function collapse.

For marching cubes, it appears chunkifying the process with multithreading was a winning approach that was superior to multithreading with a counter. Multithreading will be the right choice in many scenarios for big terrains and meshes, but it won't always be the right decision so our big takeaway is to consider the specifications of the content you are generating before making this decision for yourself in your endeavors.

References

[1] Raouf Touti. *Perlin Noise*. 2023. URL: https://rtouti.github.io/graphics/perlin-noise-algorithm.

[2] Cline Lorensen. *Marching Cubes look-up table: the 256 different configurations of vertex cell polarity are generalized by 15 cases due to existing rotations and symetries*. 2017. URL: https://www.researchgate.net/figure/Marching-Cubes-look-up-table-the-256-different-configurations-of-vertex-cell-polarity_fig1_277975970.

[3] Allison Casey. *Wave Function Collapse Java*. Version 1.0. Nov. 2020. URL: https://github.com/allison-casey/wavefunctioncollapse.

[4] Maxim Gumin. *Wave Function Collapse Algorithm*. Version 1.0. Sept. 2016. URL: https://github.com/mxgmn/WaveFunctionCollapse.

[5] Jan Orlowski and Amy Lee. *Parallel Wave Function Collapse*. Dec. 2019. URL: https://amylh.github.io/WaveCollapseGen/.