Parallel Wave Function Collapse

Deng-Hao Xu
NYCU 313551026
Hsinchu, Taiwan
za970120604@gmail.com

Yuan-Ching Chou
NYCU 312553016
Hsinchu, Taiwan
qaqaqqa147@gmail.com

Tsung-Wei Tsai
NYCU 313553002
Hsinchu, Taiwan
wwiigh@gmail.com

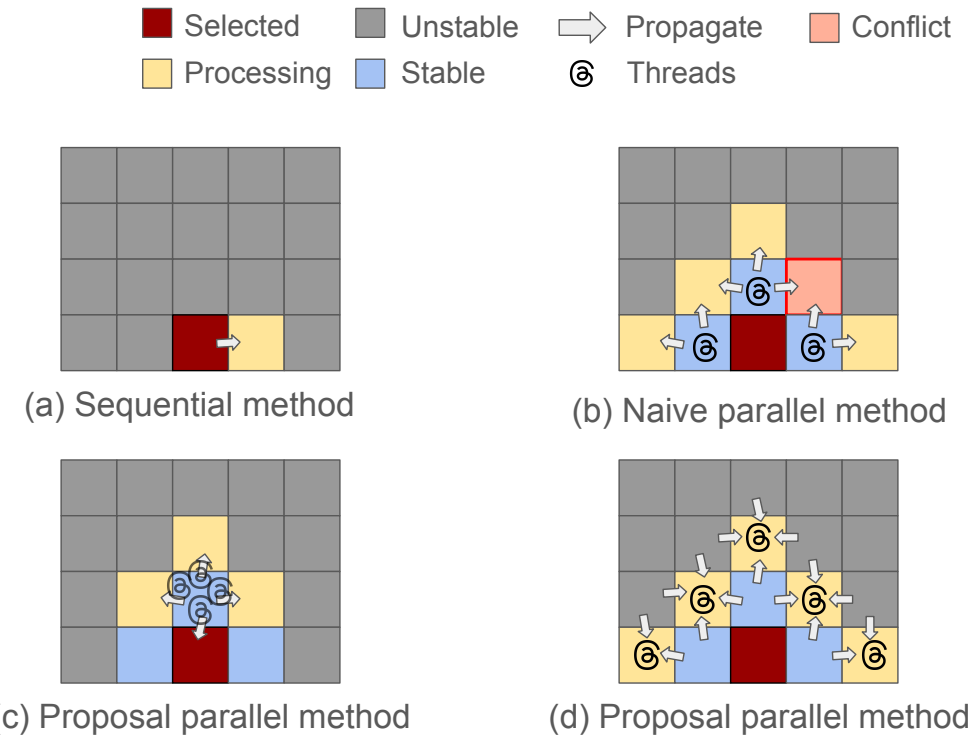

Figure 1: Four methods to implementation Wave Function Collapse (a) Sequential Method: In this approach, the propagation is performed one cell at a time in a step-by-step manner. It processes cells sequentially without leveraging parallelism. (b) Naive Parallel Method: This method attempts to parallelize the sequential approach by directly applying multiple threads. However, this can lead to write-write conflicts, where multiple threads attempt to update the same cell simultaneously, causing data inconsistency. To mitigate this, locking mechanisms are required, which introduce significant overhead and reduce the efficiency of parallelism. (c) Proposed Parallel Method 1: In this implementation(OpenMP-set), threads are assigned to propagate in four directions (up, down, left, right) from a cell. This approach distributes the workload spatially, improving efficiency while resolving write-write conflicts. (d) Proposed Parallel Method 2: In this method(OpenMP-bit, cuda), threads are assigned to process specific cells, with the propagation of neighboring cells coordinated relative to the processing cell. This effectively resolves write-write conflicts.

Abstract

In this project, we parallelize the Wave Function Collapse (WFC), an algorithm applied in unbounded, continuous map generation in games, using OpenMP and CUDA in C++.
Unlike the original WFC, which processes one grid direction at a time, parallelism allows exploration of all directions in a single call. While this reduces the time spent on each call of propagation, our experimental results show that the synchronization overhead in OpenMP and data migration overhead in CUDA outweigh the benefits, resulting in longer overall program execution time compared to the sequential version.

CCS Concepts

• **Computing methodologies** → **Texturing**; **Parallel algorithms**.

Keywords

Wave Function Collapse, Parallel computing

1 Introduction

Wave Function Collapse (WFC) is a procedural generation algorithm that is used in games and simulations to create coherent and

continuous maps or patterns. Inspired by quantum mechanics, it treats each grid cell as having multiple possible states.

1.1 Background

The algorithm details are described in Algorithm 1 below. It iteratively selects a cell with the lowest entropy (lines 7–10), collapses it to a single state (line 11), and propagates constraints to neighboring cells (lines 12–14). This process continues until all cells are resolved (lines 6, 17–21), ensuring the output adheres to predefined rules (line 4) while generating structured yet diverse content.

Algorithm 1 Wave Function Collapse Algorithm

1: **Initialize:**
2: Define the grid with $N \times N$ cells.
3: Assign each cell a list of possible states (patterns or tiles).
4: Load adjacency rules that define valid neighboring states.
5:
6: **while** there are cells with more than one possible state **do**
7: Find the cell with the **lowest entropy**.
8: **if** multiple cells have the same entropy **then**
9: Break ties randomly.
10: **end if**
11: Collapse the selected cell by randomly choosing one of its possible states.
12: **for** each neighbor of the collapsed cell **do**
13: Remove states that conflict with the collapsed cell's state.
14: **end for**
15: **end while**
16:
17: **if** all cells are collapsed **then**
18: The grid is complete.
19: **else**
20: Report a contradiction and restart or adjust.
21: **end if**

1.2 Motivation

It can be seen that this algorithm has a significant bottleneck: it requires a lot of time to converge, especially when the output size or the complexity of the input pattern increases. This limits the algorithm's efficiency and applicability to real-time applications. To the best of our knowledge, publicly available implementations of WFC, such as [1] and [2], are sequential and lack parallelized versions.

Additionally, it is clear that the entire algorithm is similar to the task of parallelizing BFS in our Assignment 3.

Furthermore, with the increasing complexity of game visuals driven by advancements in GPU technology, parallelizing WFC on both the CPU and GPU is highly meaningful and valuable.

Based on these three points, we are motivated to explore parallelization for this problem.

2 Related Work

We found [3] on the internet that also focuses on parallelizing Wave Function Collapse. They focused on parallelizing the "propagation" step, which is the bottleneck during the entire program execution.

- They did not parallelize WFC in CUDA, but we successfully implemented the CUDA version and found that reducing data migration overhead is pivotal.
- They parallelized WFC using OpenMP, which is also used in our experiment.
  - Each tile checks the states of its four neighbors independently during each round of propagation. This method is similar to our proposed OpenMP Set version.
  - Further optimization was introduced by utilizing work queues. If a tile changes, only its neighbors are affected. Instead of checking all tiles, they maintain a queue of potentially changing tiles. Whenever they process a tile and find that a change has occurred, its neighbors are added to the queue. This method is similar to our proposed OpenMP Bit version.

  They achieved a 2x–4x speedup using the first method with four neighbors, which is consistent with our experimental results. However, they did not observe any speedup with the second method utilizing a work queue due to contention between threads, which also aligns with our experimental results.
- They initially used static thread assignment in OpenMP but found that dynamic (guided) assignment slightly improved speed, suggesting that work imbalance may accumulate over many propagation rounds. This behavior was also observed in our experiment.
- They also parallelized WFC using pthreads, which we did not implement in this project. They first statically interleaved pthreads with their tiles, and in the second approach, they statically assigned pthreads to tile blocks. Their results show that the interleaved method outperformed the blocked method, and they reported that the pthread method consistently performed worse than both OpenMP methods.

3 Terminology

Here we define two terms **Iteration** and **Function call** that will use in this paper.
In this paper, the term **Iteration** refers to the propagation process where cells in the current queue are processed, and new cells requiring further processing are added to the next queue.
In contrast, a **Function call** meaning the propagation process starting from a selected cell. Propagate recursively until the state of all the cells in the grid becomes stable. The experimental results (Table 1) report the number of function calls and their execution times, rather than focusing on the number of iterations within each propagation function call.

4 Proposed Solution

We use OpenMP and Cuda in C++ to implement parallel version of Wave Function Collapse.

4.1 State recording

We implemented two methods to track the possible states of a cell: The first method uses std::set to record the state of each cell, while the second method uses a 64-bit unsigned long long (ull) for recording. These two concepts are applied not only in the sequential method but also in the OpenMP (OpenMP-Set and OpenMP-Bit) and CUDA methods. When reducing the states of a cell based on its surrounding cells during propagation, we use set intersection for the first method, and a bitwise AND operation between two unsigned long long integers for the second method.

4.2 Sequential Baseline

The sequential method, which is our baseline, simply follows the algorithm flow mentioned above. Since we have two ways to record the possible states of a cell, we also have two sequential methods called Naive and Bit, and their details are provided in Section 5.

4.3 OpenMP

We propose two OpenMP-based methods to optimize propagation.

4.3.1 Set-Based Method. Threads are assigned to propagate in one of four directions (Figure 1.c). This assignment minimizes write-write and read-write conflicts. Each thread collects all valid superpositions in its assigned direction, then propagates the valid state to neighboring cells using an intersection operator. If a neighbor's state changes due to propagation, it is added to the queue for processing in the next iteration.

4.3.2 Bit-Based Method. Propagation is treated as a Breadth-First Search (BFS) algorithm (Figure 1.d). Threads are assigned to cells in the queue. Instead of propagating states in four directions, the method gathers valid states from neighbors and removes invalid states for the cell being processed. This approach effectively avoids write-write conflicts. However, this method still encounters read-write conflicts, which are resolved in the next iteration without affecting the behavior of the WFC algorithm. In our experiments, we demonstrate and ensure that all methods produce the same grid state after each function call.

We do not use a BFS-like approach to Set-based method, as it would require locking each cell, introducing significant overhead.

4.4 Cuda

In Cuda, we propose one method to optimize propagation.
In the propagation stage, we assign one thread to each cell in the grid (Figure 1.d). Each thread will collect state information from neighbors, and use these information to update self state. Once every thread is stable, which means no cell will update the state, we can stop the calculation and finish the propagation stage.

5 Experimental Methodology

We initially had four datasets: Summer, Road, Example, and RPGMap (Figure 4). We chose the Summer dataset for the experiment because it has the most complex rules and yields the most stable results. The parameter settings are random seed 999 and a grid size of 64 × 256 (Height × Width). For the analysis, we ran five different versions of the experiment.

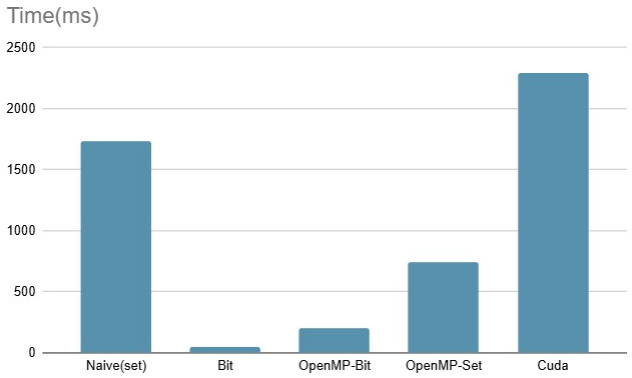

Figure 2: The total execution time of five methods.

- Naive: The baseline implementation, follow the method in Figure 1.a, use std::set<int> to store cell state. Each integer in the set means one possible solution in this cell. We say the propagate is stable means every cell's set is not changed.
- Bit: Method same with Naive, but use one uint64 to store cell state. We assume every cell will not have over 64 possible solutions, so we can use one uint64 to express the set<int> in the Naive version.
- OpenMP Bit
- OpenMP Set
- Cuda: Base on bit version, convert to cuda version. In cuda version, we use 1024 threads in 1 block, total have (Height × Width)/1024 blocks, assume (Height × Width) is divisible to 1024.

All the results are run on the class workstation. The output and time result of this setting can be seen in Figure 2,3 and Table 1.

6 Experimental Results

6.1 OpenMP

In the set-based method, we achieve a 2x speedup compared to the naive implementation. However, a theoretically 4x speedup is not attainable due to the characteristics of the WFC algorithm. At the start of the algorithm, many cells require propagation, allowing for effective parallel processing. As the algorithm progresses, most cells collapse into specific states that no longer need processing or propagation. It takes little time compared to the beginning of the algorithm. This limits the parallel processing, moreover it still incurs parallel overhead, such as enqueue fetch_and_add operations. For the parallel bit-based method, the performance is 3.7 times slower than the sequential bit-based version. This is because the parallel overhead outweighs the actual propagation time in most iterations. One propagation function call itself takes less than 0.013 ms on average, while the overhead is approximately 0.015 ms, making the parallelization inefficient in this case.

Table 1: The experiment time result of five different methods. Propagate Total/Collapse/SelectCell means total time spend on Propagate/Collapse/SelectCell stage, Propagate Max/Min/Avg means Maximum/Minimum/Average time spend on propagate in one function call. Total means time to finish whole algorithm.

(ms)	Function call count	Naive	Bit	OpenMP Bit	OpenMP Set	Cuda
Propagate Total	4232	1732.77	53.956	204.06	747.905	2295.14
Propagate Max		4.5	0.134	0.336	2.84	1.416
Propagate Min		0.0001	0.0001	0.015	0.01	0.261
Propagate Avg		0.409	0.013	0.048	0.177	0.545
Collapse	4232	0.586	0.265	0.434	0.766	0.382
SelectCell	4232	0.198	0.12	0.354	0.353	0.202
Total	1	1736.23	56.643	208.117	752.772	2299.34

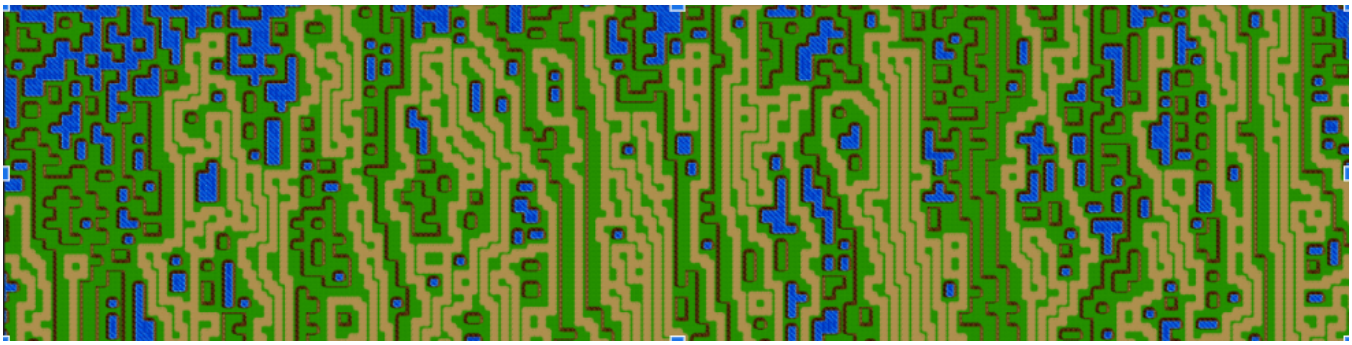

Figure 3: Result of experiment with Summer Dataset, random seed 999, grid size 64×256(Height × Width)

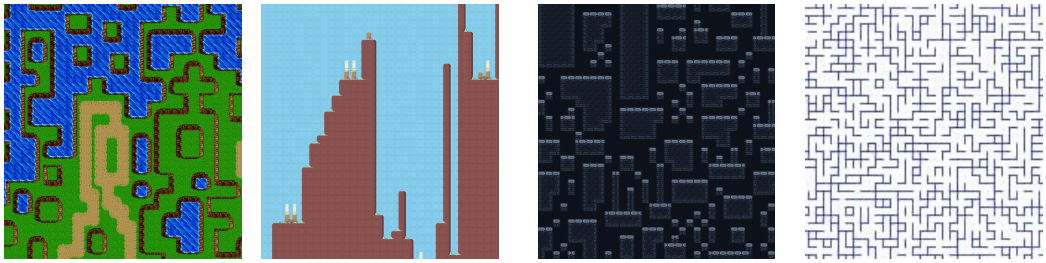

Figure 4: Other result. (a) Summer Dataset (b) Example dataset (c) RPGMap dataset (d) Road dataset

6.2 Cuda

In the cuda version, we can speedup the performance in small grid, for example, 16 × 32. But the speedup result is not good in large grid like Summer dataset, it is about 1.32 times slower than the naive version, and 40.59 times slower than the sequential bit-based version. We found some issues that caused the poor performance.

6.2.1 Data transfer between device and host. Every time we start a kernel function, we need to transfer grid data from cpu to gpu, it is a main bottleneck in our cuda version. We can see in the Table 1, the max propagate time is 1.42 in cuda, which is better than Naive version. This means our cuda version has speedup in the first few propagate, where many cells need to update the state. But in min propagate time, cuda is slower than Naive, this is because we need to handle data consistency between gpu and cpu, which is time consuming.

6.2.2 Global memory access problem. In our implementation, we use global memory to store each cell's state, resulting in a long memory access time. Because we need access and change the cell state frequency, and the new state must be seen to every thread, which makes it difficult to use local memory to speedup the memory access.

6.2.3 Kernel function call too frequency. We need to call a kernel function in every Propagate stage, which means when our grid size is big, we need a frequency call kernel function to finish our task.

In our experiment, we call the kernel function 4232 times, which will take some overhead.

7 Conclusions

Throughout this project, we identified several key findings:

- The sequential bit version is too fast to be surpassed by either OpenMP or CUDA.
- Compared to the sequential set (naive) version, the OpenMP Set version achieves a 2x speedup in overall execution time, while the OpenMP Bit version is 3.7x slower than the sequential bit version.
- Compared to the sequential set (naive) version, the CUDA version achieves multiple-fold speedup for small grid sizes (e.g., 16×32) but fails to scale effectively for larger grids (e.g., 64×256).
- These results highlight the importance of addressing synchronization overhead and optimizing data transfer between host and device in CUDA when applying parallelization techniques for different problem sizes.

Our future research prospects focus on two main areas:

The first is to address synchronization overhead. Future research could explore new parallelization strategies or utilize OpenMP Tasks for greater efficiency.

The second is to minimize memory transfer latency in CUDA. Advanced features such as Pinned Host Memory, Asynchronous Memory Copy, and CUDA Streams could be leveraged to achieve this.

References

[1] Oleg Klimov Emil Ernerfeldt, Nick Renieris. 2016. *wfc.* https://github.com/emilk/wfc [Online; accessed 12-October-2024].
[2] Maxim Gumin. 2016. *Wave Function Collapse Algorithm.* https://github.com/mxgmn/WaveFunctionCollapse [Online; accessed 10-October-2024].
[3] Amy Lee Jan Orlowski. 2019. *WaveCollapseGen.* https://github.com/amylh/WaveCollapseGen [Online; accessed 12-October-2024].