PCP Assignment 1 Report

MNGKHO012

Parallelism Implementation

After examining the serial implementation of the Abelian Sandpile problem, I identified the update method in the Grid class as a significant performance issue. This method iterates through every cell in the grid sequentially, processing each cell one after another in a single thread. For large grids, this sequential processing is time-consuming and inefficient, presenting a prime opportunity for parallelization to improve performance.

GridWorker Class and compute() Method

To parallelize the update method, I created a GridWorker class that extends the RecursiveTask class. The GridWorker class has the **compute()** method which is responsible for performing the work of the update method but allowing for parallelism.

The compute() method works with the grid array, it checks if the size of the grid is smaller than the sequential cutoff, if it is, the method processes the grid sequentially, if it is not then it is too large to be processed sequentially and must be split up. If the grid is too large the method determines whether to split the grid vertically or horizontally based on its dimensions. This is useful if it is split multiple times because after it is split the first time it is no longer a square and hence the program needs to determine the most relevant way to split it to maximize efficiency.

The left task, which is an instance of the GridWorker is then forked using .fork() to run asynchronously. This means that it will start executing in a separate thread managed by the Fork/Join framework. The compute method executes the right task in the current thread, in this way for all instances both left and right are being executed in parallel by different threads. The join method is then used to ensure that the right task waits for the left task to finish execution after which their results can be combined. The compute method returns a Boolean that signifies a change, if either the left or right task observes this change, it will be returned.

This divide-and-conquer approach leverages multiple CPU cores to perform computations simultaneously, thereby potentially reducing the total computation time.

Fork/Join Framework

In the Grid class, a ForkJoinPool is created using the commonPool method, which manages a default number of worker threads. The update method is modified to use the invoke method for parallel computation. This method submits a task to the pool and ensures all parts are completed before returning the result, effectively distributing tasks across available CPU cores for efficient resource utilization**.**

Sequential Cutoff

The sequential cutoff is used as a metric to determine the base case of the compute method. The cutoff determines the smallest size of the subarray/sub-grid that will be processed sequentially rather than in parallel. For small tasks, the overhead of managing tasks can outweigh the benefits, making sequential processing more efficient. The cutoff value is dynamically adjusted based on the number of available CPU cores. $CUTOFF = 65 * \frac{numcores}{4}$
This formula was chosen through empirical testing to balance the benefits of parallelism with the overhead of task creation. The cutoff value scales with the number of cores to ensure efficient task distribution and minimize processing time for sub-tasks that are small enough to be handled sequentially.

Validation

The parallel algorithm's correctness was validated by comparing its output with the serial algorithm's results across various input sizes and initial conditions. This process involved verifying that the final stable state, where all cells stabilize with no cell exceeding four grains, and the number of timesteps required to reach this stability were consistent with the serial implementation. This means that the parallel algorithm produces the same results as the serial version, proving its accuracy. Therefore, the parallel approach can be considered a reliable and correct method for simulating the Abelian Sandpile model. A second metric of checking validity was the images produced, the parallel images were compared to the serial to confirm they were congruent, hence ensuring correctness.

Benchmarking

The parallel algorithm was evaluated on two systems: a Windows-based computer with 6 cores and a Senior Lab machine with 4 cores. Tests were conducted for various grid sizes, including 8x8, 16x16, 50x50, 65x65, 100x100, 200x200, 400x400, 600x600, 800x800, and 1000x1000. Each grid size was run 5 times on each system, and the average execution times were recorded to ensure reliability and consistency. This approach helped assess the algorithm's performance and scalability across different hardware configurations and provided a clearer understanding of its efficiency in utilizing available computational resources.

Senior Lab Machine	**6 Core Windows-based Machine**
- CPU: 12th Gen Intel® Core™ i3-12100 - Cores: 4 - Threads per core: 2 - logical processors 8	- CPU: 12th Gen Intel® Core™ i5-12400 - Cores: 6 - Threads per core: 2 - Logical Processors: 12

Problems Encountered

1. Finding the best sequential cutoff or even an appropriate one required extensive testing to minimize the overhead from task creation and synchronization.
2. Handling and processing large data files requires careful management to avoid memory overflow and ensure efficient computation.
3. Processing large files placed significant demands on system resources. This made testing time-consuming and occasionally affected overall system performance, impacting the efficiency of the tests.
4. My computer lacked the power and performance specifications to effectively test the algorithm. It also experienced overheating issues during tests, making it impractical to use in this project.

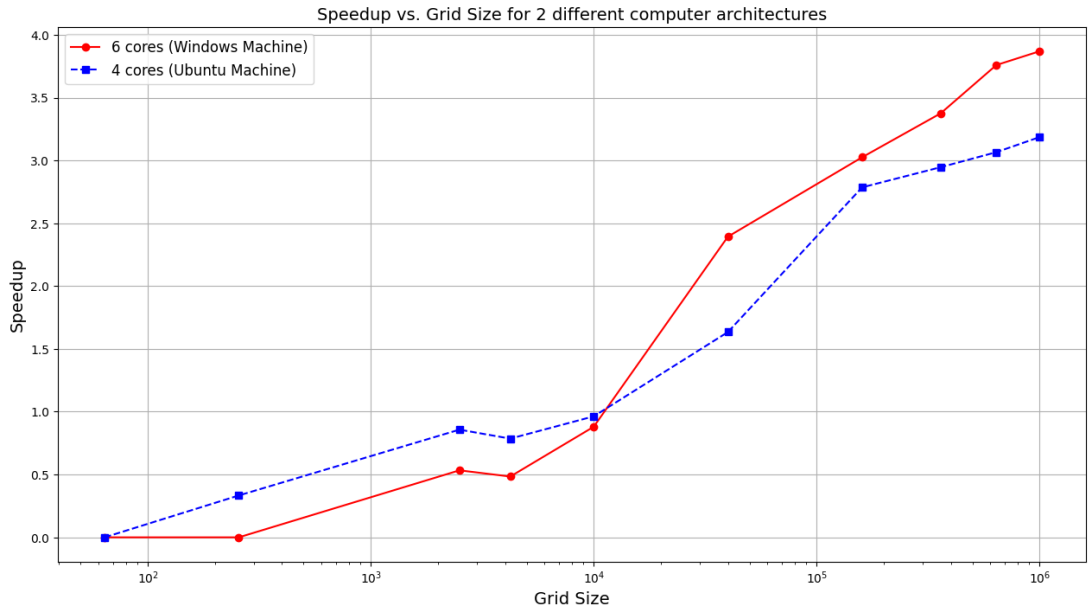
Results

To evaluate the performance of the parallel algorithm, I compared its execution time against the serial approach for varying grid sizes. The speedup for each grid size was calculated using the formula:

$$SpeedUp = \frac{Time_{Serial}}{Time_{Parallel}}$$

Where $TimeSerial$ is the execution time of the serial approach and $TimeParallel$ is the execution time of the parallel approach.

Using Python's Matplotlib and pandas libraries, I created graphs to visualize how speedup varies with grid size and across different architectures.



Discussion of Results

From the analysis of the execution times, it is evident that speedup surpasses 1 when the grid size exceeds 100x100, indicating that the parallel algorithm becomes more efficient than the serial approach for larger grid sizes. Initially, the serial algorithm is faster than the parallel one due to the overhead associated with implementing parallelism. This overhead can include the time required for managing threads and synchronizing tasks, which can outweigh the benefits of parallelization for smaller problems. However, as the grid size increases beyond 100x100, the benefits of parallelism become more pronounced. From this point onward, speedup values consistently exceed one, demonstrating that parallelism is advantageous for larger datasets.

The graph illustrates a clear upward trend in speedup from grid sizes of 100x100 to 1000x1000, suggesting that the parallel approach scales effectively with problem size. Although the rate of change in speedup is not perfectly linear, the general trend indicates that the parallel algorithm performs increasingly better as the grid size grows. This improvement in performance scaling reflects the fact that while the serial algorithm's execution time increases with larger problem sizes, the parallel

algorithm becomes more efficient, leading to higher speedups and making parallelism a preferred choice for handling larger problems.

The 6-core machine consistently outperforms the 4-core machine in terms of speedup for the larger grid sizes. For instance, the best-observed speedup on the 6-core machine is **3.87** at a 1000x1000 grid size, while the 4-core machine achieves a lower maximum speedup of **3.19**. Ideally, with perfect parallel efficiency, the speedup should be proportional to the number of cores, which means a 6-core machine could achieve up to 6x speedup and a 4-core machine up to 4x. The observed speedups fall short of these ideal values, but the 6-core machine still demonstrates better performance, reflecting its greater capacity to handle parallel tasks effectively compared to the 4-core machine. The 6-core machine achieves higher speedups because it can better utilize parallel processing, by distributing the workload amongst multiple processors.

Spikes and Anomalies

It was observed that the 4-core machine exhibits higher speedup than the 6-core machine for grid sizes smaller than 100x100. This is due to the higher overhead of managing parallel tasks on the 6-core machine, which can outweigh the benefits for smaller grids

There is also noise in the graph of the speedup because sometimes the fluctuations in system load can introduce variability.

Conclusions

While the parallel implementation did not achieve the ideal speedup, it proved to be worthwhile. It consistently showed improved performance with larger grid sizes. This trend indicates that the parallel algorithm is well suited for large datasets, as it can utilize multiple CPU cores to optimize computation. This fast processing is crucial for simulations like the Automaton Simulation which tend to be of a large scale and would be cumbersome if run sequentially. Another key takeaway from this is that optimizing factors such as the sequential cutoff and hardware architecture could further enhance the performance of parallel algorithms, making them highly valuable for computationally intensive tasks.