CSC2002S Assignment PCP1 2024

A logo of a university

Description automatically generated

**Student number:** PDXORO001 **Course:** CSC2002S

**Name:** ORORISENG **Surname:** PAADI

**Parallel Programming with Java: Parallelizing an Abelian Sandpile Simulation**

**Introduction**

This experiment aims to demonstrate the performance difference between sequential and parallel programs, this report will investigate the speed differences between the two types, the relative performance boost, the limit of that speed boost and benchmarking the parallel program to determine under which conditions parallelization is worth the extra effort involved.

The algorithms used in questions were the AutomatonSimulation Serial which we were given and the AutomatonSimulation Parallel I made.

**Methods**

*Approach*

The parallel code utilizes the Fork/Join framework in Java to efficiently update a grid using a divide-and-conquer algorithm. Initially, a `ForkJoinPool` is created to manage the execution of tasks, and a `GridUpdateTask` is defined to handle the grid's sub-portions. The task is submitted to the pool, and its result is awaited. The `compute()` method of `GridUpdateTask` checks if the subgrid is small enough to be processed directly, in which case it updates the grid sequentially. If the subgrid is too large, it is divided into three smaller tasks that are forked and submitted to the pool using `invokeAll()`. The `join()` method is used to wait for the completion of these tasks and combine their results to determine if any grid cells were updated. This divide-and-conquer approach ensures that the workload is evenly distributed among available processor cores, leveraging parallel execution to reduce overall computation time significantly compared to a sequential solution.

*Validation and Benchmarking*

The validation of the parallel algorithm involved several steps to ensure its correctness, consistency and performance improvements. To ensure that the parallel algorithm produced the same results as the serial solution. I first ran the serial code using the input files provided, and recorded the *number* of time steps it took to reach a stable state as well as the *time* it took to reach that state, and then used the same input files on my parallel solution and recorded the number of time steps to reach a stable state. Both the serial and the parallel algorithm produced the same results when given a certain grid size. The final grid states from the serial and parallel solutions were compared for a range of data sizes (19 different grid sizes were used), both small and large grid sizes were tested to ensure the algorithms worked correctly for various scales, the results were the same for all tested data sizes, confirming that the parallel algorithm correctly replicated the serial algorithm’s functionality

To verify that the parallel algorithm produced consistent results across multiple executions, I ran it multiple times with the same initial grid configuration. The final grid state from each run was compared to ensure that the results were identical across all executions. The results were consistent across multiple runs, indicating that the parallel algorithm produced stable and repeatable outcomes

Furthermore, to validate the parallel algorithms improved performance over the serial algorithm, I observed both the execution times of serial and parallel solutions for a range of grid sizes. This included small, medium and large grid sizes to evaluate the scalability of the parallel algorithm. The parallel algorithm demonstrated a significant reduction in execution time compared to the serial algorithm, especially for larger grid sizes

Machine architectures used:

|  |  |
| --- | --- |
| Machine 1 | Machine 2 |
| CPU: Intel Core i7-9700k (8 cores) | CPU: Intel Core i5-1135G7 (4 cores) |
| RAM: 16 GB | RAM: 8 GB |
| Operating System: Windows 10 | Operating System: Windows 10 |

Table 1

Table 2

**Discussion**

Range of the data sizes for which the best speed up is obtained

The best speedup is observed for medium-sized grids, specifically around the specifically around the 230x 230 grid size. For this grid size, the speedup is approximately 2.33 times. This indicates that the parallel algorithm is most efficient for these data sizes.

I observed that medium-sized grids strike a balance between task creation overhead and the efficiency gained from parallel execution. Smaller grids do not benefit significantly from parallelization due to the overhead task management and the relative simplicity of the computation.

For very large grids, while there is substantial computation to be parallelized, the overhead of managing a very high number of tasks, combined with potential memory and resource contention, slightly reduces the efficiency.

For medium-sized grids, the workload is large enough to keep all cores busy without the overhead of task management becoming a significant bottleneck. Each core can process a huge amount of data without frequent interruptions or idle periods.

How the max speedup compares to the ideal for each architecture

For a 4-core architecture, the ideal speed would be 4 times if the program were fully parallelizable. However, the maximum speedup for the 4-core architecture, as shown in Table 2 is around 2.33 times for the 230 x 230 grid. The observed speedup is approximately 58% of the ideal speedup. This shows that while the parallel algorithm effectively uses the multiple cores, there are overheads and inefficiencies stopping it from reaching the full potential speedup. These could include the aforementioned task management overhead, synchronization delays, and memory access contention.

Moreover, the ideal speedup for an 8-core architecture would be 8 times under perfect parallelization conditions. The maximum speedup observed for the 8-core architecture is again around 2.33 times for the 230 x 230 grid size. The observed speedup is about 29% of the ideal speedup. The lower percentage compared to the 4-core architecture suggests that the overheads and the inefficiencies have a more pronounced impact as the number of cores increase. Perhaps the higher the number of cores introduces additional complexity in task management and synchronization, and the potential for resource contention becomes big

Spikes/Anomalies/Trends

Finding the ideal threshold was a huge difficulty for me, I found that the threshold affected the execution time. My parallel program would give me the same results as the serial solution but the time taken for the parallel solution would be slower than the serial solution which also affected speedup, so I had to fiddle around a bit with the threshold, to get reasonable results, and because of that it really messed up the speedup, it was literally giving me abnormal values, for example, I was getting speedups that are significantly greater than the ideal speedup (I was getting speedups in the 20s range)

For smaller grid sizes, the speedup remains constant at 1. This suggests that the overhead of task creation and management outweighs the benefits of parallel execution for small data sizes. The small size of the task doesn’t allow for effective utilization of multiple cores

There is noticeable increase in speedup starting from the 16 x16 grid size. For example, the speedup jumps significantly for the 16 x16 and 24 x 24 grid sizes. This indicates that the grid size is now large enough for the parallel algorithm to effectively divide the tasks among the available cores, reducing the impact of overhead.

For larger grid sizes, the speedup fluctuates. For instance, there is a noticeable drop in speedup for the 53 x 53 grid size compared to the 45 x 45 grid size. Maybe, such fluctuations can be linked to the variability in task synchronization and resource contention, as well as the specific characteristics of the data and how it is divided among the tasks.

For very large grids, the speedup tends to plateau or increase very slowly. This could be due to the increasing overhead of managing a lot of tasks and the higher likelihood of resource contention as all cores are heavily utilized.

**Conclusion**

When measuring the performance of a parallel program, the primary performance metric is runtime. Speed and other factors such as scalability are important only to the extent that they result in improved run time.

Multithreading, when done correctly, generally makes your program run faster especially for larger amounts of dat. This was shown through my investigations on how speedup changes with different grid sizes.

As seen in *Graph 1*, as the grid sizes increase so does the speedup until it reaches a peak. From then on there is no considerable change in speedup regardless of how much the data input sizes increases. However, it is still faster than the sequential solution in 4 core and 8 core machines. It is also understood that the parallel program plateaus because it reaches the processing limit of the cores available.

Finally, for smaller grid sizes a parallel program is not needed as it may only just increase the runtime rather than improve it. Smaller grid sizes had worse or even no improvement. This could be because the Java Fork/Join Framework utilizes some sequential parts and parallel parts before initializing the invoke method therefore for small operations it becomes costly rather than helpful