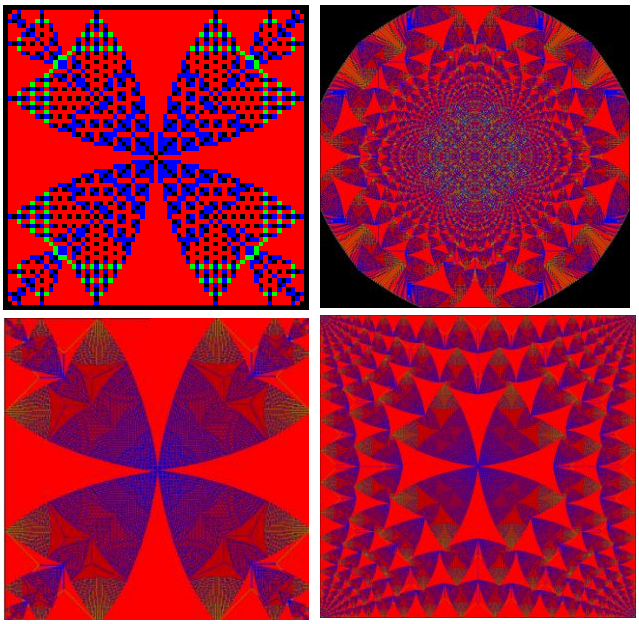
CSC2000S
PCP1 Assignment 2024
Parallelising an Abelian Sandpile



KAVYA KAUSHIK
KSHKAV001

[1]

[1] All output files are provided as images in the output folder along with all the grid csv files in the input folder. All parallel generated images are given as imagename.png and serial generated images are names imagename_serial.png. Keep in mind for certain grid sizes different cell configurations are tested.
GitHub repository link: https://github.com/kavyaKaushik510/PCP_1_Abelian-Sandpile.git
Code in branch Parallel_1

Introduction and Background

The problem outlined in this assignment is that of a two-dimensional cellular automaton called an **Abelian Sandpile.** This automation is a grid where each cell contains values representing "grains of sand". When the number of grains in a cell exceeds 4, in the next time step the cell distributes the grains evenly to the neighbouring cells. The border around a grid is called a "sink".

A starting configuration is created of the Abelian Sandpile where each cell is given a value and it is changed based on the logic described until it reaches a stable state where the grains in each cell are less than 4. The final state is represented by different colours for different values in the cell (black for 0, green for 1, blue for 2 and red for 3). The serial program to simulate this has been provided in the Grid and AutomatonSimulation classes.

This assignment aims to develop a parallel solution for the problem. The parallel solution is required to achieve the following:

- Runs faster than the serial solution to solve the slow runtime especially when larger grids are processed.
- Avoid all data races
- Only utilise the fork/join framework for synchronisation.
- Provide accurate outputs from the specified input format (.csv files) and validate the output(.png images).
- The solution must be run on at least two machines and the outputs must be thoroughly evaluated.

Solution description

Parallelizing Approach

The approach used in the solution was to recursively divide the grid into four segments until the sequential cutoff is reached. A **ParallelAutomatonSImulation** class is created which reads the grid from the input. This class makes a call to the method in the **Parallel Grid** class to execute the parallel algorithm. The Parallel grid class extends the Grid class and contains an inner class to execute the parallel algorithm. This class creates a fork join framework and makes a call to parallelise the algorithm as described above where the grid is recursively divided into 4 segments. The parallel algorithm is called on each segment and the join framework is used to ensure synchronisation. Once the grid reaches a size smaller than the set threshold the grid is processed as in the given serial solution and is updated accordingly.

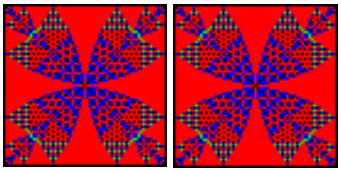
Apart from implementing an effective parallel approach the code also uses an optimised method to update the grid where just the references of the grid and the grid it is updated from using a **swap grids()** method. This is done to avoid recursively using an array which takes longer.

The **sequential cutoff** is determined after extensive testing between the values of 55 ranging up to 5000. Dynamic cutoff techniques are implemented as well to test the best sequential cutoff. After thorough analysis a sequential cutoff of 2000 is set.

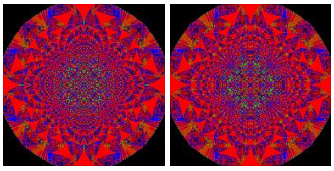
Solution Validation

The solution is validated by comparing the outputs obtained for each input using the parallel program with the serial program. The output images are compared using image comparison software online and the processed grids are printed out and compared to ensure that they are equal. Images from the parallel output for a few sample grid inputs can be found in the output folder of the code. The number of stable states required to achieve the output is also compared between the serial and parallel program to verify that the output is valid. Different cell configurations (all cells 4, all cells 8, centre value only).

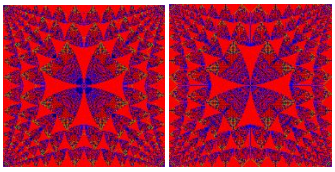
Sample Output



65 x 65 grid size
Number of steps to state: 1156



517 x 517 grid size
Number of steps to state: 98904



1001 x 1001 grid size
Number of steps to state: 471192

Benchmarking using different machine architectures

- Optimisation methods to ensure battery optimisation and to ensure that the machine is not running unwanted tasks are implemented for effective testing.

Two machine architectures are used to benchmark the program for thorough analysis.

1) **ASUS VivoBook Flip 14** (my personal laptop)
    - Machine Specifications:
        - Processor: 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz
        - Base Speed: 2.242 GHz
        - Cores: 4; Logical processors: 8

Table 1 shows the runtimes for the serial and parallel algorithms on the 4-core machine

Input size (rows x columns)	Parallel Solution Time(ms)	Serial Solution Time(ms)	Speedup
8 x 8	8	0	0
16 x 16	8	0	0
65 x 65	98	36	0.4
200 x 200	1752	1124	0.6
517 x 517	43971	80080	1.8
720x720	108963	209634	1.9
920x920	493431	1162573	2.4
1001 x 1001	543785	1588794	2.9

2) Senior Lab Computers: sl-dual-287

- Machine Specifications:
    - Processor: 12th Gen Intel® Core™ i3-12100 x 8
    - Cores: 8

Input size (rows x columns)	Parallel Solution Time(ms)	Serial Solution Time(ms)	Speedup
8 x 8	2	1	0.5
16 x 16	3	2	0.7
65 x 65	39	31	0.8
200 x 200	724	989	1.4
517 x 517	24138	80867	3.4
720x720	46490	138539	3.0
920 x 920	320597	1109131	3.5
1001 x 1001	427647	1506082	3.5

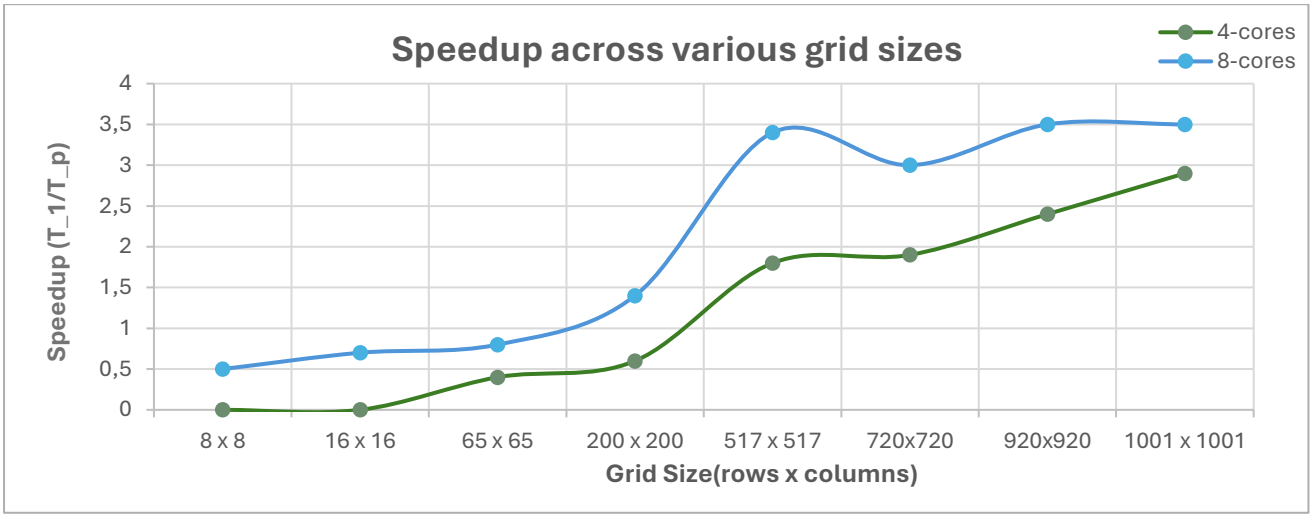
Problems & Difficulties

Testing to set an efficient sequential cutoff that gives a good runtime for a large range of grid sizes was a very time-consuming process. The process was optimised by implementing a totalgridsize/2 formula and then the constant threshold values were further optimized from there. The coding process was quite intensive to ensure that all functions are thoroughly optimised to ensure minimum time overhead.

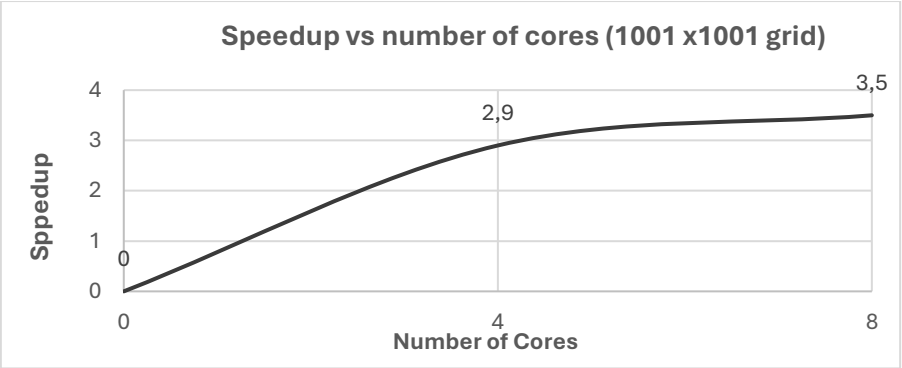
Results – Speedup Graphs

Speedup is calculated as follows, $Speedup = \frac{T_1}{T_p}$ where T_1 is the time taken by a single processor in the serial program and T_p is the time taken by the parallel program when process on a machine with p number of cores. The graphs below are used to analyse the performance of the parallel algorithm.

Graph 1 shows the speedup obtained for each of the grid sizes using the parallel algorithm



Graph 2 shows the speedup obtained for the 1001x1001 grid on both the 4 cores and 8 core machine



Discussion

The program performed as expected. The overall parallel algorithm performed faster than the serial algorithm for larger grids and slower or similar for smaller grids. This validates the logic that parallel algorithms are only beneficial for larger programs. This is because creating parallel threads creates a lot of time overhead and this overhead is only beneficial if the program is quite computationally heavy. The measurements are reliable since the solution is validated in terms of the image outputs and the number of stable states. The runtimes are obtained as the mean values run for multiple tests for each grid and the runtime values were quite close which shows the stability of the algorithm. It is also seen that different cell configurations have different number of process (a big centre cell value takes long to process). It can be seen that the overall speedup is much higher for an 8-core machine and it also starts speeding up on a smaller grid as compared to the 4-core machine. It should also be noted that each amhcine is also running other background process, hence ideal speedup is very difficult to achieve.

4-core-laptop: The parallel algorithm starts speeding up after grid size 517x517. This is as expected and as explained above. The ideal expected speedup for this machine is x4 and the maximum speedup obtained is x2.8 on the 1001x1001x grid which is reasonable. The program runs without any anomalies.

8-core computer: The parallel algorithm starts speeding up after grid size 200x200. This is as expected since parallelization works better on larger grids. The maximum expected speedup on this machine is x8 and the maximum speedup obtained is x3.5 on the 1001x1001x grid. It can be noticed that the 720x720 grid has a smaller speedup however this is because it works on quite a computationally heavy grid with a large value only in its centre so it can be justified.

Conclusion

In conclusion the assignment achieved its objective by providing a parallel algorithm for the Abelian Sandpile which is both faster than the serial program (for larger grids) and correctly produces the required output with the specified input. The parallel algorithm is quite beneficial to speedup this problem by giving a speedup as high as 3.5 on an 8-core machine and it is expected that much larger grids would produce even high speedups. The higher computational power the machine has with a greater number of cores the faster the parallel algorithm will perform. **In conclusion, it is beneficial to use parallelisation(multithreading) in this context since it significantly speeds up the Abelian Sandpile algorithm.**