CSC2002S Assignment PCP1 2024

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

My parallelization approach uses the Fork/join Framework like a mechanism to manage parallel tasks efficiently. The main class '**AutomatonSimulation'** initiates the simulation by reading the grid from a CSV file and repeatedly updating it until a stable state is achieved. The **'Grid'** class handles the grid and its updates using the Fork/Join Framework to parallelize the computation. The '**Grid.update()'** method invokes a '**ForkJoinPool'** to manage the parallel execution, creating instances of the **'GridTask'** class to update portions of the grid. The divide-and-conquer algorithm works like this: '**GridTask'** recursively splits the grid into smaller portions until they are small enough to process sequentially. If a portion is below the **'CUTOFF'** size, it is updated directly; otherwise, it is further divided either by rows or columns, and new '**GridTask'** instances are created for these portions. These tasks are then executed in parallel using the '**fork()'**, and then their results are combined using '**join()'**. This approach efficiently leverages multiple processors to improve performance, particularly for large grids, by parallelizing grid update computations

Validation and Benchmarking

In order to validate my parallel algorithm to make sure it was correct, consistent and improved performance, 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 as well as the time. Both the serial and the parallel algorithm produced the same number of time steps when given a

certain grid size, but different computational times as expected. The final grid states from the serial and parallel solutions were compared using a range of different grid sizes (19 different grid sizes were used, see Table 1 and Table 2), both small, medium and large grid sizes were tested to ensure the algorithms worked correctly for various scales.

To verify that the parallel algorithm produced consistent results across multiple executions, I ran it multiple times with the same initial grid configuration. For example, I would take a 120 by 120 grid input file, put it in the command and run the same command 8 times, the results were the same, but obviously giving different times but not too different from each other, usually a 10-millisecond difference, and sometimes I would get the same computational time even when running the same command multiple times. 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, when observing both the execution times of serial and parallel solutions for a range of grid sizes the parallel algorithm demonstrated a significant reduction in execution time compared to the serial algorithm, particularly for larger grid sizes, but was not so efficient for smaller grids (see Table 1 and Table 2).

This part of the experiment was done using two different machines, one with 8 cores and the other with 4 cores and I used the '**Tick()'** and '**Tock()'** methods to get the computational time and the integer variable '**counter'** to get the number of time steps. For the Table 1 and 2, the speedups were rounded off to one decimal place

Machine architectures used:

Machine 1	Machine 2
Senior Lab Computer (8 cores)	My Laptop (4 cores)
Operating System: Linux (ubuntu)	Operating System: Windows 11

Table 1: 4 core machine

Grid Size		Number of TimeSteps	Time (ms)		Speedup
Rows	Columns		Serial Time	Parallel Time	
8	8	18	1	3	0,3
10	10	56	1	3	0,3
16	16	67	1	4	0,3
24	24	279	8	14	0,6
32	32	565	18	28	0,6
45	45	1145	33	41	0,8
53	53	1394	38	48	0,8
65	65	1156	41	50	0,8
74	74	2589	71	91	0,8
89	89	2221	77	94	0,8
97	97	4101	175	167	1
120	120	7221	323	310	1
230	230	14074	2708	1436	1,9

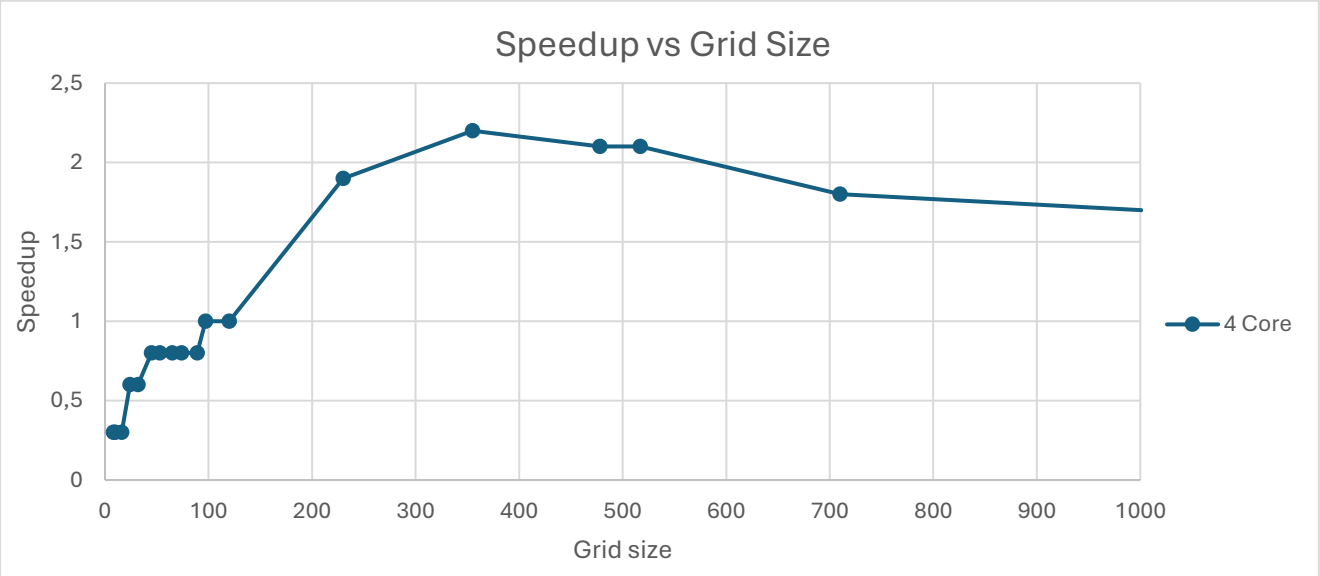
355	355	57830	27724	12663	2,2
478	478	108684	96368	46470	2,1
517	517	98904	94832	46123	2,1
710	710	134274	255846	139214	1,8
1001	1001	471192	1917028	1124146	1,7

Table 2: 8 core machine

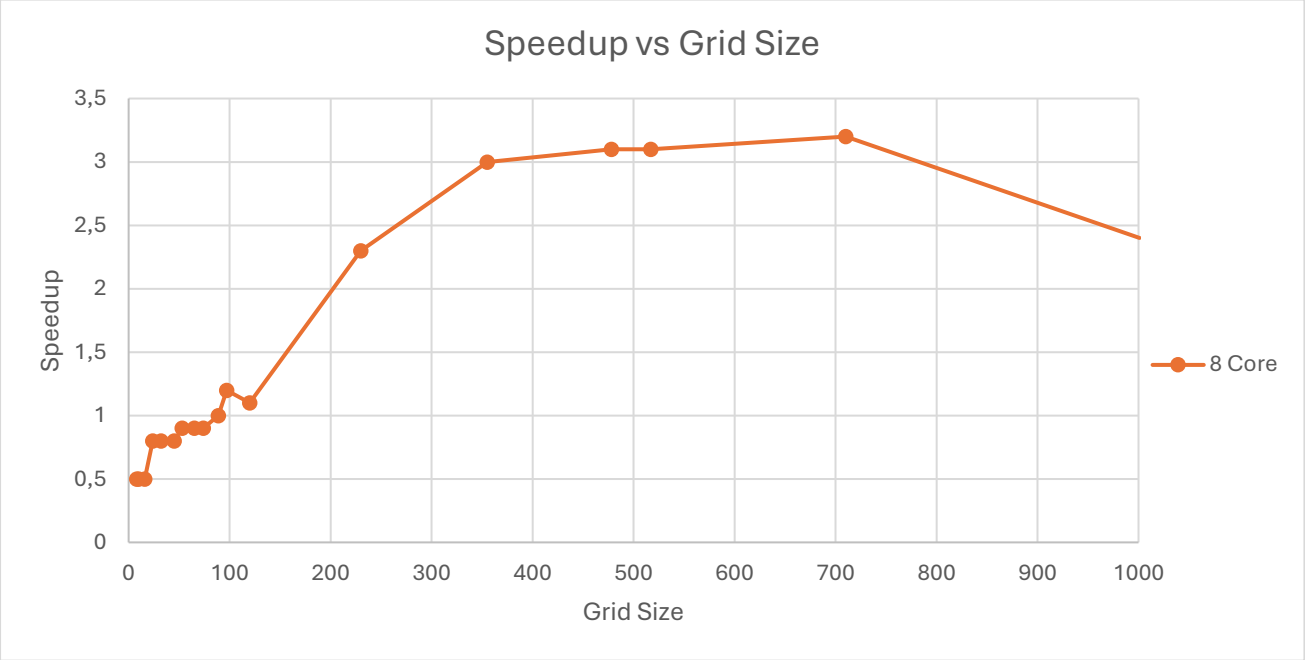
Grid Size		Number of TimeSteps	Time (ms)		Speedup
Rows	Columns		Serial Time	Parallel Time	
8	8	18	1	2	0,5
10	10	56	1	2	0,5
16	16	67	1	2	0,5
24	24	279	7	9	0,8
32	32	565	13	17	0,8
45	45	1145	29	37	0,8
53	53	1394	40	43	0,9
65	65	1156	42	45	0,9
74	74	2589	68	75	0,9
89	89	2221	68	71	1
97	97	4101	169	139	1,2
120	120	7221	285	257	1,1
230	230	14074	2363	1016	2,3
355	355	57830	24932	8401	3
478	478	108684	86409	27503	3,1
517	517	98904	85643	27954	3,1
710	710	134274	223751	70394	3,2
1001	1001	471192	1615290	676197	2,4

Discussion

Graph 1: 4 core machine speedup graph



Graph 2: 8 core machine speedup graph



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

Based on my data collection, the best speedup for both machines is achieved with large grids. For the 4-core machine, the optimal speedup of approximately 2.2 occurs at a grid size of 355x355, while the 8-core machine reaches its peak speedup of around 3.2 at a grid size of 710x710. In these ranges, the workload per core is substantial enough to minimize the impact of overhead, leading to more effective parallel processing. Smaller grid sizes result in lower speedup due to significant overhead relative to computation, whereas very large grid sizes have a decline in speedup as overhead increases, potentially due to memory access bottlenecks or inefficient load distribution. Thus, the best speedup is achieved when the grid size is large enough to utilize the cores effectively but not so large that overhead negates the benefits of parallelization.

Max speedup vs ideal speedup

For a 4-core machine, the ideal speedup would be 4. However, the observed maximum speedup is approximately 2.2, which is only about 55% of the ideal speedup and achieved with a large grid size of 230x230. Similarly for the 8-core machine, the ideal speedup is 8, but the maximum speedup observed is approximately 3.2, about 40% of the ideal, which occurs around a grid size of 355x355. Similar to the 4-core machine, the less-than-ideal speedup suggests inefficiencies that limit the effectiveness of parallelization. For example, I noticed that applications that were running/open in the background (e.g. chrome tabs, Spotify, etc) tend to have an effect on time, which end up affecting speedup since it is calculated using the serial and parallel time, and I noticed this because, when I was running the algorithm for grids that were not as large (e.g. 45x45 or 65x65), I would get times that were above 300 milliseconds, which I thought was oddly suspicious given the number of cores I was dealing with for each architecture, but as soon as I closed those background programs and ran my files again,

the time would show a significant reduction and higher speedups. Furthermore, in the initial stages of the experiment finding the ideal threshold was a challenge for me, I found that the threshold affected the time for my parallel algorithm. My parallel solution would give me the same results as the serial solution but the time observed for the parallel solution would be slower than the serial solution, so I had to fiddle around a bit with the threshold, to somewhat get reasonable times, but after fiddling around the best cutoff for me was 90. So the observed maximum speedup for both architectures being less than the ideal ones, shows that while parallelization does improve performance, it falls short of perfect efficiency. This can be because of what I mentioned before and other things like overheads from task management, synchronization, etc., which are not really accounted for in the ideal scenario. Additionally, load imbalances and limitations from Amdahl's law, which states that the non-parallelizable portion of a program can limit speedup, contribute to the reduced performance. So, while the parallel algorithm shows benefits, it does not achieve the ideal speedup because of these practical limits.

Trends, anomalies and spikes

Both graphs exhibit a general trend of increasing speedup as the grid size grows, up to a certain point. For the 4-core machine, the speedup increases until it peaks at the grid size of 355x355, after which it begins to decline. A similar trend is observed for the 8-core machine, where the speedup increases until it peaks at a grid size of 710x710. This suggests that larger grid sizes initially benefit from parallelization, as the workload is divided among more cores, leading to better utilization. However, beyond these grid sizes, the trend shows a decline in speedup, maybe because of increasing communication overhead or other factors that reduce parallel efficiency.

Moreover, an anomaly observed in the data is the decrease in speedup observed in both architectures after their respective peaks. I expected the speedup to increase even more as the grid size kept scaling up, but for the 4-core machine, the speedup started to drop after the grid size of 355x355 as seen in Table 1 and Graph 1, and for the 8-core machine, the decline began after the grid size of 710x710 as seen in Table 2 and Graph 2. Another thing is the relatively low initial speedups for both machines, particularly at smaller grid sizes, where the speedup is well below 1, indicating that parallelization is not very effective at these smaller scales. These anomalies perhaps indicate that there are overheads or bottlenecks in parallel execution such as inefficient load balancing or excessive communication, which become more pronounced as the grid size either grows too large or too small for effective parallelization.

As seen from Graph 1 and Graph 2, huge variations of speedup (spikes) are not really present, but instead the graphs show that for certain ranges of grid sizes the speedup is of a certain value and continues to stay at that value (plateaus) and then picks up a little bit. For example, Graph 2 and Table 2 show that for the grid sizes 8x8, 10x10 and 16x16, the speedup stays as 0.5 for all three grids and then slightly increases and stays as 0.8 for the grid sizes 24x24, 32x32 and 45 x45, then increases and stays as 0.9 for the grid sizes 53x53, 65x65 and 74x74. For the grids that come after the 74x74 grid, only then does the speedup show differing values for each grid. Perhaps these plateaus for

certain ranges like the one where the speedup is 0.5 for all three grids, can be linked to the fact that the variation in grid scaling is not too much, so the workload size for grids in a certain range is more of less the same, hence resulting in the same speedup.

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 data. This was shown through my investigations on how speedup changes with different grid sizes.

As seen in *Graph 1* and *Graph 2*, 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 increase. However, it is still faster than the sequential solution in 4 core and 8 core machines, particularly for larger grids. 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.