Zolile Zoko ZKXZOL001
CSC2002S

Parallel and Concurrent Programming Assignment 1

Introduction and methods

Divide and conquer is a strategy used to tackle complex problems by breaking them down into smaller, more manageable subproblems. These subproblems are then solved individually, and their solutions are combined to solve the original problem. In this assignment, the divide and conquer approach was used to increase the efficiency of an Abelian Sandpile simulation by converting the provided existing serial programs into parallelized versions that are both efficient and correct. This approach was implemented using the Fork/Join Framework, which allows a problem to be divided into subproblems based on a chosen sequential cutoff and then the subproblems are assigned to workpool of threads that tackle the problem in parallel and after computation is done all the individual results are added up. The sequential cutoff is the problem size at which further division of the problem into smaller subproblems no longer provides a benefit, as the overhead of additional task division outweighs the efficiency gains.

Two Java classes were provided: *AutomatonSandpile***,** containing the main method for running the simulation, extracting grid dimensions from a CSV file, and creating the grid object; and *Grid*, which stored grid parameters and methods for updating the grid based on Abelian Sandpile rules, tracking timesteps, and generating a PNG image of the final grid state.

The focus of parallelization was on the Grid class, renamed *ParallelGrid* in the parallel solution. An inner class, *GridTask*, was added to handle the divide-and-conquer approach using the Fork/Join framework. The update method in *ParallelGrid* was modified to create a *GridTask* object, invoke a ForkJoinPool, and update the simulation's timestep as needed.

The compute method, placed in *GridTask*, handled the recursive breakdown of the problem until a base condition was met. If the condition was met, the grid was computed directly; otherwise, the problem was halved, with one task assigned to the main thread and the other forked. The main thread invoked the compute method, while the forked thread used join to make the main thread wait until it completed its assigned task. A sequential cutoff, determined by problem size, available cores, and a minimum cutoff, controlled the grid's breakdown. The larger value, determined by Math.max, was used as the sequential cutoff.

This approach was applied to all grid sizes, with data collected on output images and computation times on both a Lenovo i3 Ideapad and a Senior lab computer. The results were used to generate plots and analysis for the report,each data set for each grid was taken more than once to ensure the validity of the captured data for that specific dataset.

Validation of the algorithm

Two methods to validate the correctness of the algorithm were done, one was a visual means of validating the output images of the parallel algorithm by comparing them with the serial program and the other was done by generating hash values for both the parallel and serial program images for each grid size and then comparing the values, if the values are the same then the images are said to be the same.

The images below show outputs from the serial and parallel program and from the images below it can be seen from a visual inspection perspective that the two images are identical.

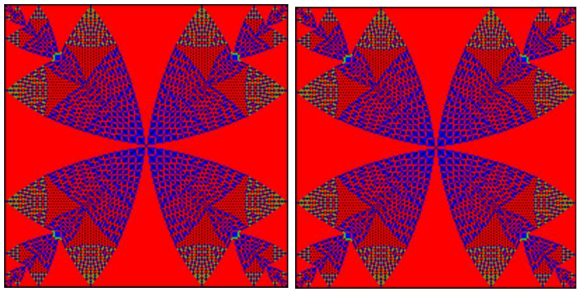

Figure 1 and 2: ouput images from the serial(right) and parallel programs(left).

Images below show the results generated when passing serial and parallel program output png images on a website called https://www.online-convert.com/ that can generate hash values from images and from the results below it can be seen that the hash values for the output images for 200 by 200 grid are the same as a result the two images are the same.

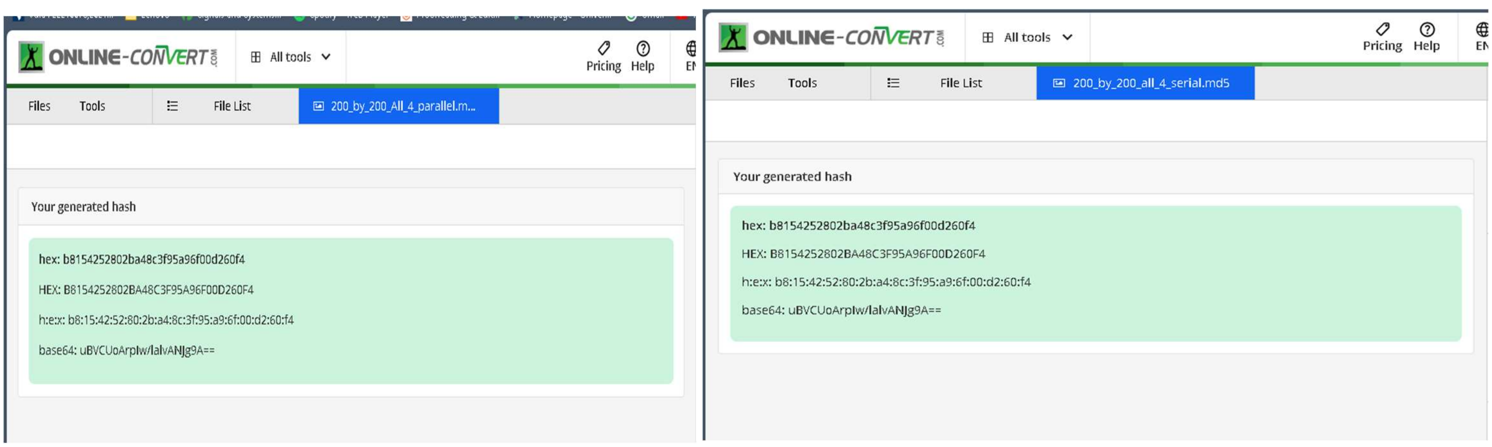

Figure 3 and 4: hash values for 200 by 200 grids from serial(right) and parallel program(left).

Based on the hash values and visual inspection of output images it can be concluded that the parallel program solution is correct.

Benchmarking

The standard that the parallel program solution needs to meet or surpass is set by the serial program, so the performance of the parallel program was benchmarked against that of the serial program. The performance of both was tested on two different machines, a Lenovo IdeaPad i3 and a senior lab pc, both machines are 4 core machines. The dataset used ranged from an 8 by 8 grid to a 1001 by 1001 grid. Each grid was ran 2 to 5 times on each machine for both the serial and parallel programs and the time taken to reach stable state was taken as data to be used to calculate the respective speedup on the two respective machines by dividing the time taken by the serial program over the time taken by the parallel program, a value greater than 1 for speedup for a specific grid indicates that the parallel program performed better than the serial program for that specific grid size.

Datasets

Parallel Program dataset grids(PC)	s1	s2	s3	s4	s5	Average(ms)	Speedup
517	66623	69219	64136	61784	63235	64999.4	1.841999
800	263893	291146	289244	293700	280061	283608.8	1.811351
400	19169	19435	19467	20004	19825	19580	1.589658
65	94	79	79	79	81	82.4	0.575243
200	2698	2971	3185	2824	2966	2928.8	0.57696
600	83838	86314	92514	83590	95215	88294.2	1.750942
1001	1406585	1437861				1422223	1.667831
900	460387	458581	455196			458054.6667	1.795575
100	266	220	220	251	236	238.6	0.637888
300	7027	6524	6369	7248	6744	6782.4	1.990348

Figure 5: Dataset for Parallel Program testing on Personal PC.

Parallel Program dataset grids(lak	s1	s2	s3	s4	s5	Average(ms)	Speedup
517	45272	45434	44826	46456	45670	45531.6	1.777021
800	155826	185629	187270	192652	190395	182354.4	1.71356
600	58600	60560	58976	58991	60499	59525.2	1.703373
65	42	45	40	44	38	41.8	0.684211
200	1221	1251	1250	1250	1270	1248.4	1.018584
400	20046	20008	20034	19992	20015	20019	0.636166
1001	453482	443344	443698			446841.3333	3.437912
900	161459	168045	161601	161626		163182.75	3.132541
300	4072	4089	4161	4015	4114	4090.2	1.537138
100	116	122	110	106	116	114	0.673684
8	3	2	2	1	1	1.8	0
16	3	4	5	3	3	3.6	0.277778

Figure 6: Dataset for Parallel Program testing on Senior lab Computer.

Serial Program dataset grids(PC)	s1	s2	s3	s4	s5	Average(ms)
517	118356	118534	118257	124880	118617	119728.8
800	628037	484216	481389	483864	491070	513715.2
400	32786	31130	30269	30317		31125.5
65	48	48	47	63	31	47.4
200	1701	1695	1679	1695	1679	1689.8
600	155342	155452	153000			154598
1001	2249209	2479016	2387858			2372027.667
900	802238	842705				822471.5
100	157	172	173	133	126	152.2
300	9886	10057	10120	10435		13499.33333

Figure 7: Dataset for serial Program testing on Personal PC.

Serial Program dataset grids(Lab)	s1	s2	s3	s4	s5	Average(ms)
517	81108	80910	80721	80934	80880	80910.6
800	310228	320530	310113	310240	311265	312475.2
600	101314	101396	101345	101316	101597	101393.6
65	28	28	28	27	32	28.6
200	1274	1269	1273	1271	1271	1271.6
400	12611	12642	12693	12612	13119	12735.4
1001	1546457	1532862	1529284			1536201
900	511482	511825	510223			511176.6667
100	75	75	80	77	77	76.8
300	6293	6285	6292	6288	6278	6287.2
8	0	0	0	0	0	0
16	1	1	1	1	1	1

Figure 8: Dataset for serial Program testing on lab computer.

Discussion

The following figure provides speedup plots for the results taken on a 4 core Lenovo IdeaPad i3 and a 4 core Samsung Senior lab computer, the problem size in this case is the number rows that need to be divided to solve the problem of updating the grid more efficiently.

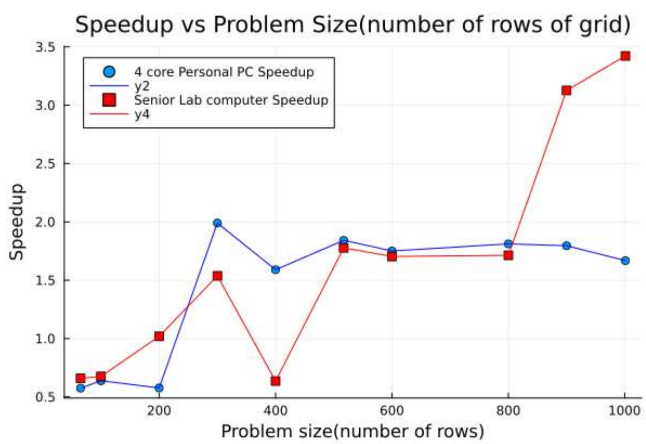

Figure 5: Speedup vs Problem size graph for two different machines.

The algorithm demonstrated consistent speedups for datasets with more than 200 rows, particularly from 200 to 1001 rows on both machines with noticeable spikes at specific grid sizes. For datasets with 8 to 200 rows, both the personal PC and the lab computer's respective performances lagged, likely due to thread creation overhead. Between 200 and 400 there was a spike that led to no speedups for the algorithm when computed in the lab computed, this is likely due to the sequential cutoff selected for that grid size causing overheads in thread creation as the speedup of the algorithm also decreased on my personal computer in this region too. Between 400 and 800 rows, both machines performed similarly, with speedups ranging between 1.5 and 2. For datasets between 800 and 1001 rows, the lab PC's speedup increased from 1.5 at 800 rows to 3.5 at 1001 rows, likely due to a more optimal sequential cutoff. In contrast, the personal PC's performance dipped from just above 1.5 at 800 rows to nearly 1 at 900 rows before recovering to above 1.5 at 1001 rows. This decline suggests that the personal PC struggled with larger problem sizes due to its lower computing power.

Despite both machines being 4-core systems with the same dynamic sequential cutoff, the lab PC, with its 4.3 GHz processor, outperformed the personal PC, which has a 1.10 GHz processor, particularly on larger datasets. This explains the lab PC's higher speedups at larger grid sizes, while the personal PC's performance deteriorated. The spike at 300 rows on the personal PC may indicate an optimal sequential cutoff that minimized overhead, leading to the best speedup for that machine. In contrast, the lab PC's performance benefited more from its higher clock speed, which allowed better handling of larger grid sizes when the sequential cutoff was appropriately chosen.

At 1001 rows, the lab PC achieved a speedup of 3.5, slightly below the expected speedup of 4 for a 4-core machine. This shortfall is likely due to background processes consuming some of the PC's computing power, preventing the ideal conditions necessary for maximum speedup.

In the case of the personal PC, the best speedup achieved was close to 2, which is half of the expected ideal value of 4. This indicates that although the algorithm provided speedups on the personal PC, its performance was poor.

Conclusion

Based on the analysis of the results from testing the parallel program solution for the Abelian Sandpile problem, it can be concluded that implementing a parallel solution is worthwhile. Parallelization reduces the time required for computation on a given problem, and its efficiency improves as the problem size increases. Therefore, it is beneficial to parallelize large problem sizes. However, the advantages of parallelism are less apparent for small problem sizes, where the overhead of creating threads outweighs the gains.

One caveat to implementing a parallel solution is that its performance may vary depending on the environment. As observed from the results on two different machines, the same algorithm does not always provide the same benefits across different platforms. Consequently, a parallel solution should either be multiplatform, meaning it delivers consistent efficiency regardless of the machine, or customizable, meaning that with minor modifications, it can achieve similar efficiency on different systems.