**Assignment One:**

**Parallelising An Abelian Sandpile Simulation**

*By Génevieve Chikwanha*

**Introduction**

Parallelisation has been gaining traction over the past few years due to the increased need to process larger amounts of data and the limitations of doing so on a single processor, as detailed by Moore’s law. Still, parallelisation is not a perfect solution for every problem. In fact, as I aim to show in this report, parallelisation is usually only worth it when the problem size is sufficiently large enough, otherwise, a serial approach may be better warranted. Furthermore, I also aim to show that Gustafson’s law seems to be correct in claiming that the parallel portion of a program (Tp) dominates over the serial portion (Ts) when the problem size and processors increase, which supports my prior thesis.

For this experiment, I converted a serial Abelian Sandpile Simulation into a parallel program where threads worked to separately update each row according to the program rules (detailed in the assignment brief). I detail my methods and findings below.

**Methods**

**Parallelisation Approach**

About Parallelisation

To parallelise this algorithm, I used the popular and highly effective Java ForkJoin Framework with the common pool implementation. This synchronisation framework works by splitting the problem into smaller tasks that are then put in a “work pool”. Each thread can take tasks from the workpool when they are able to do so. The number of threads created is determined by the Executor, but since I used the *.commonPool()* method, I can assume that the number of threads created is likely equal to the number of cores on my laptop – 1, which gives me:

*10 – 1 = 9 threads*

My Parallelisation Approach

I focused on parallelising the .*update()* method since its time complexity in the serial version is roughly O(n2) per timestep, which spells trouble for larger inputs. So, each grid is divided into rows with a sequential cutoff of 25. This cutoff was determined by taking the smallest input size that displayed any significant speedup from parallelisation, which in this case was the 517 by 517 grid, and experimenting to find the lowest cutoff that guaranteed the greatest speed. After much testing, I found that 25 was not only the lowest cutoff, but it was also the best cutoff overall. Higher cutoffs, even for much higher input sizes, like the 1001 by 1001, made the program run slightly slower.

Optimisations

One optimisation that helped to give me significantly better results was to use the double buffering technique. This is a technique primarily used in computer graphics to make drawing images (or rendering game states) as seamless as possible (Nystrom, 2014). To explain this in context, the technique creates two copies of the current state and then switches between them. The copying and swapping operation usually has a performance cost, however, since we are only swapping references to the arrays (since arrays are pass-by-reference) and not the actual content of the array, the overhead is negligible.

**Architectures**

This program was run on two computers, namely:

* An Apple MacBook Pro M2 with 512GB SSD, 16GB RAM, and 10 cores, 6 for performance and 4 for efficiency.
* Xxx

The results were uniform across the architectures.

**Benchmark and Validation**

Benchmark

To benchmark and validate this program, I created a Python script (attached as ProgramRunner.py) which ran both programs one after the other for a total of 10 trials (per architecture). The script captures the time (in milliseconds) and steps taken per trial and puts the results into an Excel document (results.xlsx) (which you can find attached to this assignment folder). It also captures the minimum time out of every single trial for both programs so that the minimum time can be compared and thus benchmarked. This also means that since the program is run 20 times in total on each input size, we have a better guarantee that the results are uniform and replicable. The input sizes were as follows:

* 8 by 8
* 16 by 16
* 32 by 32
* 65 by 65
* 125 by 125
* 250 by 250
* 517 by 517
* 750 by 750
* 1001 by 1001
* 2000 by 2000

Validation

Furthermore, to validate the correctness of the parallel program, the Python script outputs each version’s final grid into a .csv file and then compares each .csv file line by line to ensure that they are equal. The script then outputs a correctness score to gauge how often the parallel grid matches the serial grid. As you can see in the results document, the correctness score is 100%, meaning the parallel program runs correctly 100% of the time.

Replication

To replicate both the validation and benchmarking, the same Python script can be used or copied. All that will be needed are the input files, the Makefile, and a similar folder structure (or edit the script to accommodate the user’s folder structure). All of these are provided in this assignment folder. The user can then simply run Python ProgramRunner.py in their terminal, and everything should work as expected.

**Problems/Difficulties Encountered**

* **Speedup for smaller input:** Getting an increased speedup for the smaller inputs was very challenging. The parallel version of my code actually caused the total time to increase for all inputs below XXX. This was difficult to in XXX.
* **Double Buffering as necessary optimisation:** The program was running in an infinite loop, because although the grid was being updated, the xxx. I found that implementing the Double Buffering Technique helped not only to solve the issue but also to optimise the program further.

**Results**

Graphs

Discussion

Grid Size Range

The program seems to work very well when the grid size is xxx if and only if the input size is sufficiently large. Otherwise, the cutoff needs to be larger than or equal to the input size.

The parallel version performed very similarly on smaller inputs, such as the 8 by 8 or the 16 by 16. The run time was either equal or negligibly larger, such as 1 ms larger. The parallel version performed slightly worse (by less than 1 ms) when the cutoff was lower. When the cutoff was set to more than the input value (for example 10 for the 8 by 8 or 20 for the 16 by 16), the best run time was achieved, i.e. equivalent to the sequential version, as you can see in the screenshots below. This is because the parallel program is now running sequentially since one thread is doing all the work because of the comparatively large cutoff.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure 1: Screenshot of the 16 by 16 input run on the serial version of the program, which ran for 2ms.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure 2: Screenshot of the 16 by 16 input run on the parallel version of the program. A minimum run time of 2 ms was achieved, similar to the serial version.

As the input size increases to the mid-ranges (65 by 65 to 200 by 200), however, we can see these negligible differences become more noticeable. The parallel version runs almost 2x slower than the serial version. As I illustrated above, when the cutoff is larger than the input, the program is essentially running sequentially, so it performs very similarly to the sequential program, for obvious reasons. However, once the cutoff is lower than the input size and the else statement of the compute() method needs to be executed, the overhead of the recursive call and the creation of the threads slows the program down.

Une image contenant texte, capture d’écran, Police, noir

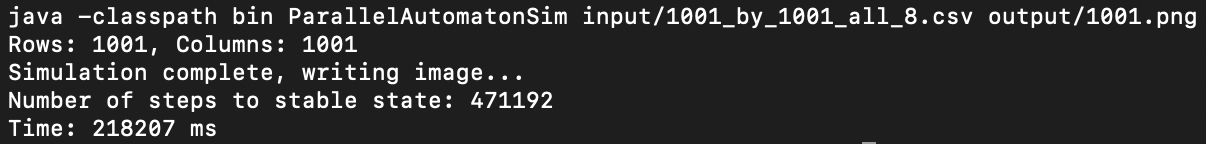
Description générée automatiquement

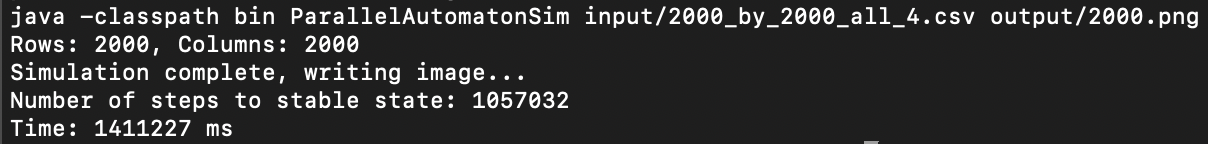
Une image contenant texte, capture d’écran, Police, noir

Description générée automatiquement

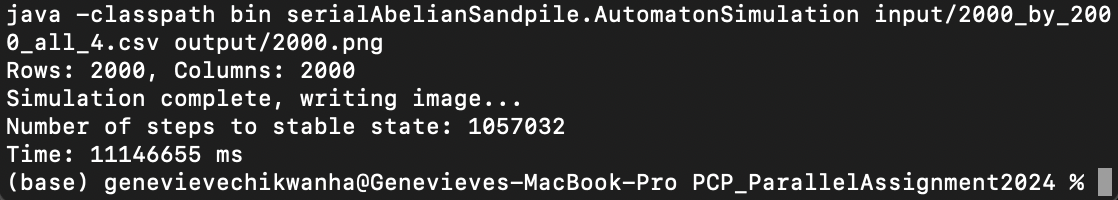
On larger inputs, like 517 by 517, the output takes much longer. In this case, it took 72 754 ms and 98 904 steps to get to a stable state. And for the 1001 by 1001 input, the program took 1 403 470 ms and 471 192 steps. This is much longer compared to a smaller input size of 65 by 65 which takes 29 ms and 1156 stable steps to run.

The parallel version ran for much longer on smaller inputs and much shorter on larger inputs. There was a 2.9x speedup for the 517 by 517 input, from 72 754 ms down to a minimum time of 20 473 ms (25 ST). Similarly, there was a 6.4x speedup for the 1001 by 1001 input which ran for 218 207 ms (about 4 minutes) compared to 1 403 470 ms (about 23 minutes) in the serial version.





23 .5 minutes



185.7 minutes

A 7.8x speedup!!!!!

Speedup

The maximum speedup achieved

Reliability

These measurements are relatively reliable since they remained consistent across the 20 trials for each input. However, as always, better reliability could be achieved by running more trials to ensure no anomalies occur. But given the limited scope and time of this assignment, the testing was rigorous enough to guarantee some sort of reliability.

Anomalies

There were no recorded anomalies in the test runs. As previously mentioned, more testing could be done to weed out outliers in the output, but for now, the results *appear* to be stable and consistent.

Conclusions

This experiment aligns with the general consensus about the effectiveness of introducing parallelism to programs. More specifically, parallelism seems to be worth it only when the problem size is sufficiently large enough. Smaller problem sizes, such as the 8 by 8 or 16 by 16 grids, benefit much more from a serial approach since they don’t have to incur the parallelisation overhead from the thread calls and synchronisation attempts. Conversely, larger problem sizes, like the 1001 by 1001 grids, benefit immensely from parallelisation.

Moreover, it is important to note that a program is influenced by many other things, such as the computer architecture, which all affect the speed of the program. This experiment has shown that, clearly, better architecture (i.e. more cores) generally has a positive effect on the speed of the program depending on the given problem size, just as Gustafson’s Law states.

There were many limitations to this experiment, namely :

* It was only tested on two architectures,
* The timesteps had to remain sequential because of their inherently sequential requirements. This limited the

However, these issues can be addressed in a more rigorous study

**Bibliography and Resources**

**Code Sources**

Double Buffering

Nystrom, R. 2014. Double Buffer in *Game Programming Patterns*. <https://gameprogrammingpatterns.com/double-buffer.html>

ProgramRunner.py

Code sourced and Adapted from the Pandas API Reference Guide.

pandas.DataFrame.to\_excel. <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_excel.html>

pandas.DataFrame.to\_csv. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to\_csv.html#pandas.DataFrame.to\_csv.

**References**

Oracle. The Java Tutorials. Double Buffering and Page Flipping

If you're looking to optimize further, you might consider:

1. Optimizing the update computation itself (e.g., loop unrolling, SIMD instructions if available).
2. Experimenting with different parallelization strategies (e.g., adjusting the granularity of tasks).
3. Using a more cache-friendly memory access pattern if working with very large grids.

TO DO:

* Make test script
  + Run both programs and find min for efficiency
  + Compare outputs for correctness
* Test on different thresholds
* Fins other optimisations

Parallelisation

* The most obvious thing to parallelise is the update method since that seems to be affecting the program the most, in terms of efficiency. The way it is written is that there are two nested for loops and two nested if statements. That makes the time complexity roughly O(n2), using the worst case ***[SHOW CALCULATION]***. Grid initialisation in the constructor should be fairly efficient, but since we are initialising the grid and the updateGrid simultaneously, and each grid is a 2D array, this operation goes from simply being a linear insert with the complexity of O(n) to being about O(n2) as well ***[SHOW CALCULATION]***. As a result, they may both need some degree of parallelisation, but the update method more so than the grid initialisation. Drawing in the board is an additional point of consideration since the program has to go to each cell and check what the value of that cell is, then assign a colour to the cell, and then it draws the image based on those values. Perhaps a more efficient library could be found or maybe this is overkill.
* Things to consider:
  + Alternative drawing tools for faster processing OR parallelising the image processing manually (though it should already be fairly quick for smaller programs)