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)

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

Serial program benchmarking

* On larger inputs, like 517 by 517, the output takes much longer. In this case, it took 72754 ms and 98904 steps to get to a stable state. And for the 1001 by 1001 input, the program took 1403470 ms and 471192 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 72754 ms down to 24091 ms. Similarly, there was a 5.6x speedup for the 1001 by 1001 input which ran for 248910 ms (about 4 minutes) compared to 1403470 ms (about 23 minutes) in the serial version.

**Methods**

**Parallelisation Approach**

To parallelise this algorithm, I used the Fork/Join Framework with the common pool implementation. This means that the number of threads created would likely be equal to the *number of cores on my laptop (10 cores) – 1 = 9*. I focused on parallelising the update() method since its time complexity in the serial version is roughly O(n2), which spells trouble for larger inputs.

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**

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.

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.

**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 only doing XXX. I found that implementing the Double Buffering Technique helped not only to solve the issue but also to optimise the program further.

**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