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**

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.

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.

**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.

Speedup

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 around 20 with smaller inputs, the best run time was achieved, i.e. equivalent to the sequential version, as you can see in the screenshots below.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure 1: Screenshot of the 16 by 16 input ran 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 ran 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, however, we can see these negligible differences become more noticeable. The parallel version runs at almost 2x slower speed than the serial version. When the cutoff is larger than the input and the program is essentially running sequentially, the program 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. Thus, parallelisation is not worth it for smaller inputs, but the opposite is true for larger inputs.

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

Description générée automatiquement

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

Description générée automatiquement

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.

Anomolies

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 seems to align with the general consensus about the effectiveness of introducing parallelism to programs. More specifically, that parallelism seems to be worth it only when the problem size is sufficiently large enough. Lower 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 x 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