PCP 1 | Parallel Abeilan Sandpile

Razeen Brey BRYRAZ002

Methods

Paralelisation Approach

- Pgrid is an adaptation of Grid and SandMain is a copy of AutomatonSimulation.
- As per the assignment requirements, the ForkJoin framework was used to handle the paralellisation.
- RecusiveAction was used to assign tasks to the thread pool. This meant that the compute() method had to be implemented.
- The approach splits the grid in halves. This is achieved by the introduction of a new constructor as shown below.

```
/**
     * Creates a subgrid of Pgrid. This is split by rows.
     * @param rowi The start row of the subgrid.
     * @param rowj The end row of the subgrid.
     * @param width The total rows from the main grid.
     * @param height The total columns from the main grid.
     * @param fg The grid component of the main grid.
     * @param uGrid The upGrid component of the main grid.
     */
    Pgrid(int rowi, int rowj, int width, int height, int[][] fg, int [][] uGrid)
```

- The compute() method splits the grids into smaller ones - by rows - using the above constructor. Once they are sufficiently small, the grid is evaluated (code from Gird's update() method is moved to compute() of Pgrid).
- The update() method of Pgrid sets effect to false then creates the main instance of the Pgrid class. This is then handed over to the ForkJoinPool object for processing. While changes are present (i.e. effect is true), the updateGrid() method is called [updateGrid() is Pgrid's copy of Grid's nextTimeStep()].
- As per the idea of a *"work stealing algorithim"* when the update() method is called, the main Pgrid object is created and handed to the ForkJoinPool object for processing. In the compute() method, the main task is subdivided into smaller tasks with fork(). Once the tasks

are of a sufficiently small size, they are evaluated with the key algorithim of the Abelian Sandpile. The threads race to complete the tasks and *steal* tasks from the bottom of the queue of busy threads. To ensure synchonisation and avoid race conditions, the join() method is used.

Optimisations

Though the PC used to write and initally test the code is a 10 core machine. The speed observed with an undefined level of parallelisim was far slower (when compared to large scale serial speeds). After experimentation, the optimal parallelism count was decided to be 4.

```
public ForkJoinPool pool = new ForkJoinPool(4);
```

Sequential Cutoff

After various trials and comparisons, the suitable value for the sequential cutoff was determined to be 12.

However, with regards to implementation, sequential cutoff was used as part of an equation to implement a variable sequential cutoff. The intention behind this is to make it scalable for larger datasets.

$$row_{end} - row_{start} < \frac{width - 2}{SequentialCutoff} - 1$$

Validation

Validation was conducted using 3 methods...

1. Visual Inspection
   - The PNG files generated by the parallel solution were compared to those from the serial solution to see if they match.
2. Comparing the number of steps taken to reach stable state.
3. Grid comparison
   - The serial and parallel solutions were modified to output a CSV file containing the final state of the grid.
   - GridValidator is a program that accepts 2 CSV files to read their arrays and determine if they are equivalent. Parallelisation is used to aid the process.

Benchmarking

- The serial program timings was used as a benchmark for the parallel program.
- It was noted that the serial program runs significantly faster on the smaller data sets and therefore, benchmarking was conducted on the 517×517 grid.
- Once a significant reduction in time was recorded, the parallel solution was deemed effective.
- To ensure proper and effective benchmarking the selection of test grids was expanded - from the given set - to include a 220 grid (with all 4s), a 335 grid (with a centre of 55000) as well as a 620 grid (all 4). These specific grids were created to cover the intermediary regions not covered by the provided grids.
- Another step was to test the program on different systems with different numbers of cores and different architectures (See *Results* section).

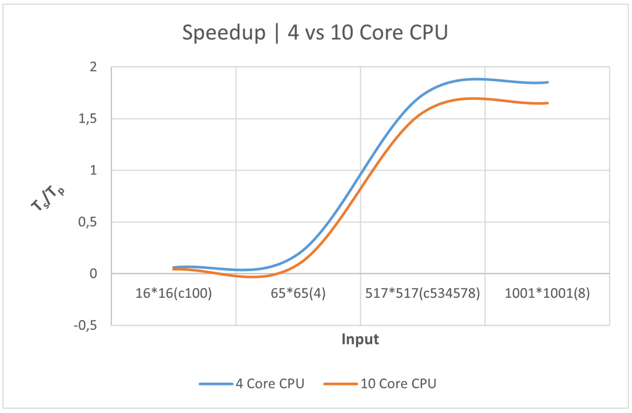
Problems

- Determining whether the program was efficient was an issue since it was only picked up after running the code on the 4-Core system that there are hardware or unrelated sotware performance issues on the 10-Core system.

Results

Testing was conducted on the following architectures.

Device	ASUS Vivobook	Lenovo Ideapad 3	Nigthmare Server
CPU	Intel i7-1255U	Intel i5-1135G7	Unknown
Cores	10	4	4
Threads	12	8	8



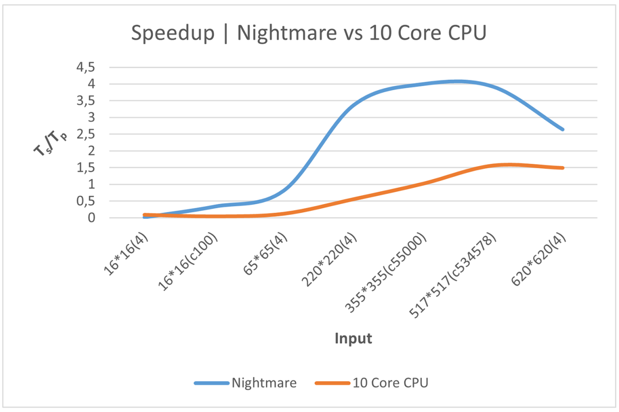


Figure 1 (Left): Speedup graph of a 4-Core vs a 10-Core system.

Figure 2 (Right): Speedup graph of Nightmare vs a 10-Core system.

Range

- Additional grids were generated to provide a wider range of observations.
- It was obeserved that the parallel program performs best at larger datasets when compared to the serial program.

Maximum Speedup

- Under optimal conditions, the maximum speedup obtained was $4,00325$ on the 355×355 grid with the Nightmare server.

$$Speedup = \frac{T_{series}}{T_{parallel}} = \frac{139041}{34732} = 4,00325$$

Reliability

To ensure reliability of the results...

- The tests were conducted 4 times per data set (620 and 1001 grid conducted twice due to time constraints) to ensure that the timings recorded were stable.
- (where possible) No other (non-essential) applications and processes were running whilst the simulations were running on the systems.

Anomalies

3 Issues/anomalies were noted in the conducting of this experiment.

1. Running both the parallel and serial programs on a 10-core machine rendered far slower results when compared to running the exact same code on an 4-core machine.

   The running conditions on both systems were kept constant (power plan set to performance etc). The parallelisation modifier of ForkJoinPool was also modified to account for the higher core machine as well as modifiying the Sequential Cutoff. None of these could bring about an increase in the performance of the 10-Core machine and it was therefore concluded that the scope of the issue was beyond the current level of (my) knowledge.

2. The Nightmare performance Spike and Dip

   As seen in Figure 2, the speedup of Nightmare dips very significantly and strongly between the 517 and 620 grid inputs. This is due to the Nightmare Server being under tremendous load stress as multiple students were connected to it running simulations. A possible reason for the spike is due to the fact that the serial and parallel solutions were evaluated on 2 separate days. This means that the performance was affected by the varying load on the server (more people on it some days than others).

3. Initially, the 10-core machine performed far slower during preliminary testing. However, the next day after the PC was shut down over night, the program has since performed at a significantly faster speed. The following were noted as possible causes...

   1. Background processes consuming resources.

   2. Thermal throttling (since the tests were done in quick succession on a laptop in an already warm enviroment [room with heater]).

Conclusion

With the research and evidence provided above it is evident that there are benefits to using Parallelism to speed up handling of tasks. This is however limited to larger datasets. In smaller datasets the overhead costs required to setup the threads take up significant amounts of time so smaller datasets are better handled serially with respect to time.

Another point to be noted from this study is that there are other factors that affect the efficiency of programs. We have seen this in the Nightmare Spike and Dip graph. Parallel programs, like serial programs, are affected by system conditions such as system load and resource allocation. To make effective use of parallel processing these factors need to be optimised.

There are also unexplained issues like the weaker performance of a 10-Core system. This however can probably be explained with a deeper understanding of computer architecture and parallel processing in Java (which I do not posess currently).

In conclusion, parallel processing is better suited for problems of a larger size and significanlty outperforms serial programs in this regard.