Program Design

The program follows the Single Program Multiple Data (SPMD) model, where a set of computation steps is applied to each element of a 2D array. The following subsections outline the key aspects of our program design:

Algorithm Overview:

For each generation, the current state of the world is first copied. The algorithm then uses a double for loop to iterate through and apply the game rules to each cell in the 2D world to generate the next state. The application of rules is done with reference to the copy of the world, while the appropriate updates to get to the next generation are applied to the original world. We also keep track of the total death toll as each new generation is generated.

Parallelization Strategy:

The parallelization strategy we ended up with is summarized by the following code snippet:

```
#pragma parallel for reduction(+:deathToll) collapse(2)
for (int x = 0; x != nRows; ++x)
for (int y = 0; y != nCols; ++y)
{
    // Apply game rules to each cell (x,y)
}
```

This strategy collapses the nested for loop and distributes the resulting workload evenly across all threads. The reduction clause causes each thread to accumulate their own partial `deathToll` result which is summed together at the end when all threads finish executing.

To arrive at this strategy we tried a few different options:

- **Critical Section vs Reduction:** performance was quite similar.
- **Static & Dynamic Scheduling with different chunk sizes**: all performed worse than the default scheduling.
  Based on the above, we decided to use reduction since the implementation is neater than critical section with OpenMP.

The `collapse(2)` clause was something we decided to use while experimenting with a world size of 50 x 60. We found that using this clause led to better performance which will be explained in the Optimization Attempt Using Collapse Directive section.

Performance Measurement

The data we collected can be found as tables in the `data.csv` file we included in our repository.

Experimental Setup

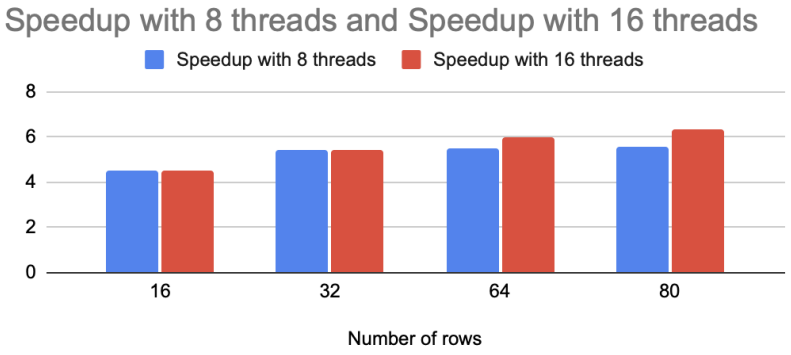
All experiments were ran on a `xs-4114` partition. We designed a total of 3 distinct experiments, each investigating the impact of different input parameters. These parameters are: number of rows, columns and generations. In each experiment, we varied one of the input parameters while keeping the other two fixed at predetermined values. We also ran each experiment with 3 different thread configurations: 1, 8 and 16 threads. Execution time and Million instructions per second (MIPS) were used as our performance metrics.

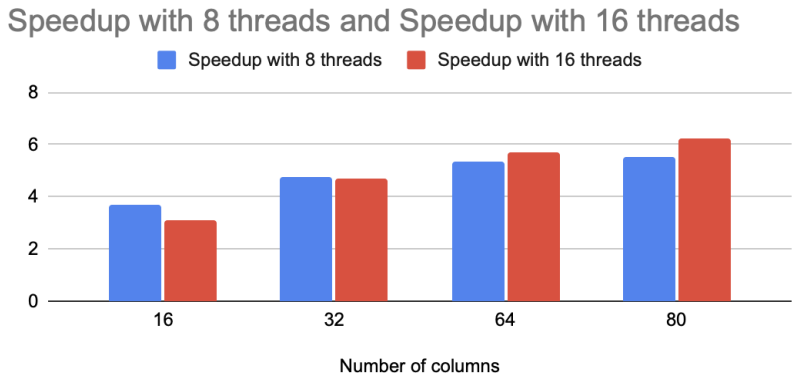
To vary the number of rows and columns, we used the input files with the naming convention: `xy_input.in`. `x` is a number and `y` is either `'r'` or `'C'` to denote row or column respectively. E.g. `80r_input.in` is the input file where the world consists of 80 rows. To vary the number of generations, we manually modified `sample_input.in`.

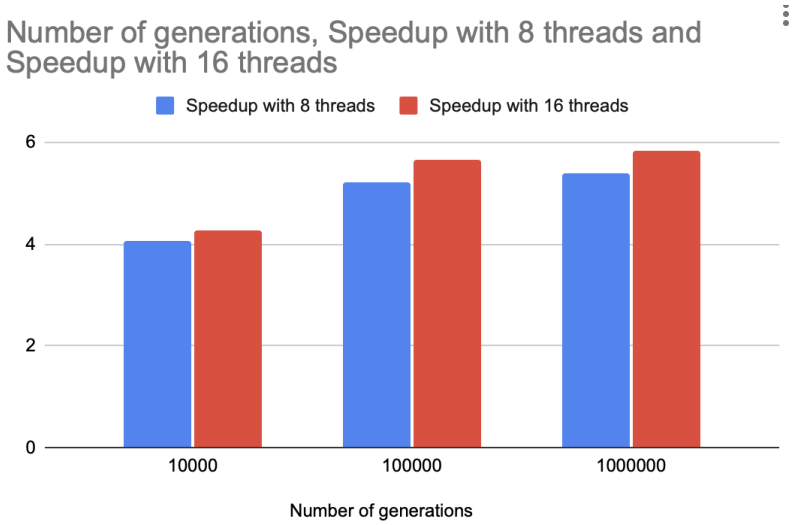
To run the program with a certain input and number of threads:
`./run_slurm.sh <input_file> <output_file> <number_of_threads>`

Results

The following charts use data collated in tables A, B and C of `data.csv` respectively.







For rows and columns, We increase them by factors of 2 to vary their input sizes. For number of generations, we increase it by factors of 10 times to vary it. As we vary the number of rows, columns and generations, the change in speedup is roughly similar. However, since the change in the number of generations is a factor of 10 compared to the factor of 2 used to vary rows and columns, we conclude that varying the number of generations affects speedup the least. On the other hand, we also conclude that varying the number of rows and columns affect speedup by roughly the same amount.

Interesting Observation on Scalability

We observed that increasing from 8 to 16 threads did not show a significant difference in speedup. This is unexpected, since we are using an architecture with 20 logical cores to run our program. Our parallelization strategy was also a simple SPMD model with no dependencies between each cell in the 2D world and thus no dependencies between threads.

We had some guesses as to why this was the case. Firstly, it could be that the sequential part of our program was limiting the speedup, following Amdahl's Law. We did see a larger difference in speedup between 8 and 16 threads when we increased the problem size (for example in tables A and B), but it did not seem significant enough to be convincing. Next, we guessed that it could be due to thread overheads and we managed to collect some data that gave us confidence in this guess. The data is in table E of `data.csv`.

To summarize our findings, there was a drastic increase in task clock and number of context switches between 8 threads and 16 threads.

Optimization Attempt: Using Collapse Directive

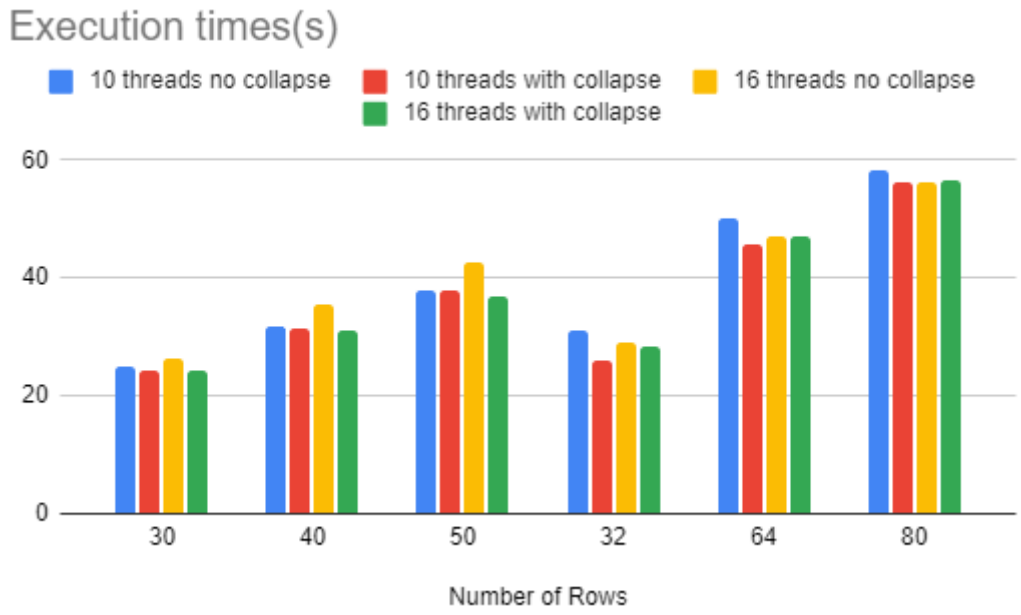
As mentioned in our Parallelization Strategy, we aimed to optimize our program by leveraging OpenMP's collapse directive to combine the double for loop into a single loop.

Initial Observation & Hypothesis

Initially, without using collapse, we observed an intriguing performance anomaly on a 50x60 world in the `sample_input.in` dataset. Surprisingly, the program ran faster with 10 threads than with 16 threads, despite utilizing the xs-4114 partition, which boasts 20 logical cores per node. Our hypothesis suggests that the row size being a multiple of 10 may have contributed to this behavior, resulting in more efficient work distribution for 10 threads.

Hypothesis Testing

To test our hypothesis, we ran our program with both 10 and 16 threads, and with varying row sizes that are either a multiple of 10 or 16. The results are summarized by the charts below and this data can be found in *Table D* of our data measurement sheet.



When the `collapse` directive is not used, the execution time with 10 threads is faster than with 16 threads when the number of rows is a multiple of 10 and not 16. Similarly, the execution time with 16 threads is faster when the number of rows is a multiple of 16 and not 10.

When the `collapse` directive is used, the execution times for 16 threads improves when number of rows is not a multiple of 16 and the execution times for 10 threads improves when the number of rows is not a multiple of 10. Furthermore, the execution times for 16 threads does not change noticeably if the number of rows is already a multiple of 16, likewise for 10 threads for number of rows that are multiples of 10. Therefore, the results seem to support our hypothesis.