Monte Carlo Minimization.

The goal of this Assignment is to programmatically compare parallel and serial algorithms on Monte Carlo Algorithm for finding the minimum of a two-dimensional mathematical function within a specified range. ForkJoin Framework is used for the parallel algorithm.

Parallelization approach.

The parallelization is achieved using Java's ForkJoin framework. The 'MonteCarloMinimizationParallel' class extends 'RecursiveTask<Integer>', which allows the program to split workload into smaller tasks that can be executed concurrently. The 'SEQUENTIAL_CUTOFF' helps avoid excessive parallel overhead for small tasks. Subtasks are forked and computed independently, and join is then used to ensure that the program waits for the subtasks to complete and combine their results correctly. Minimum search is performed by subtasks independently on the given 'searches' array. The global minimum is then calculated by comparing the minimum values from both subtasks.

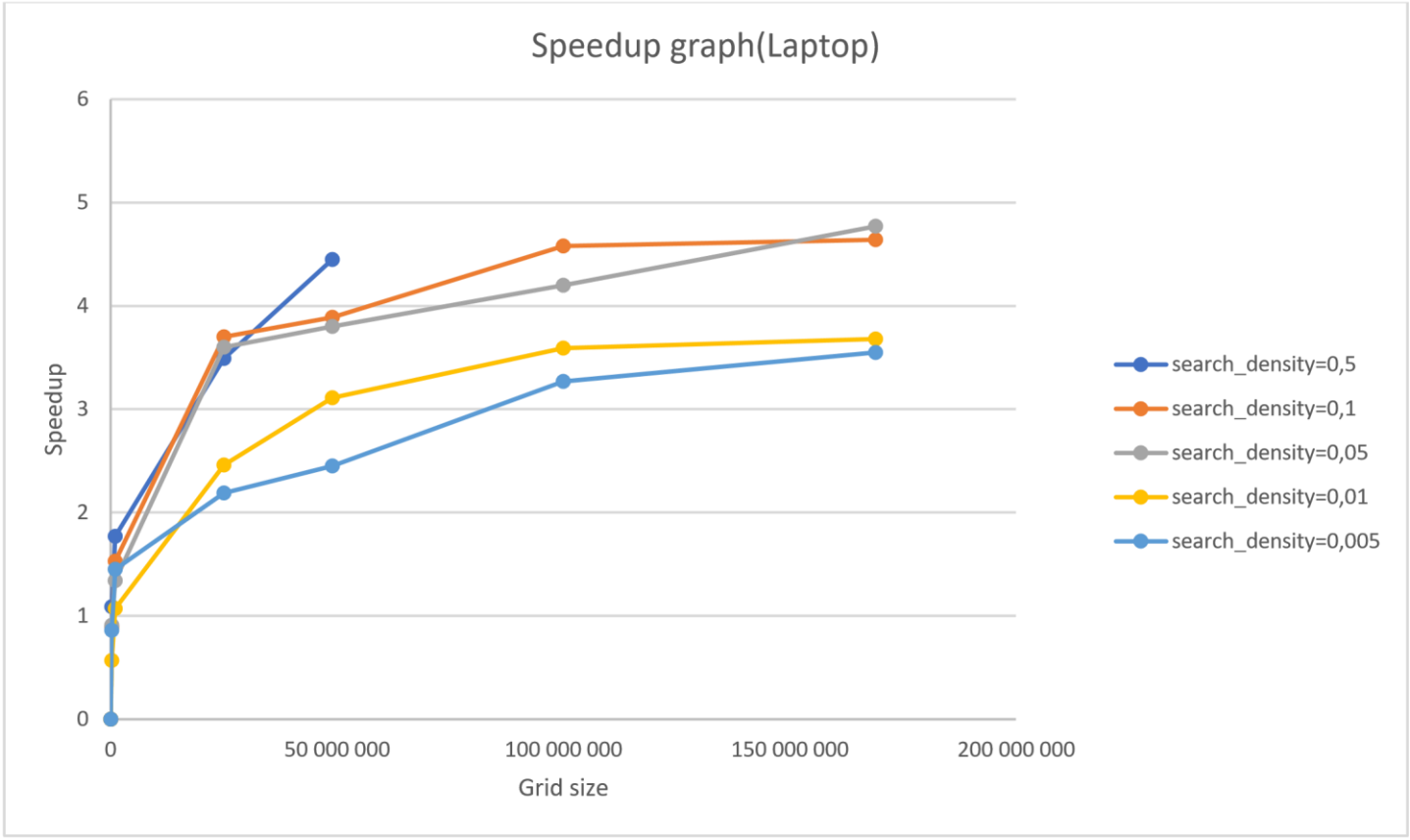
Optimisation.

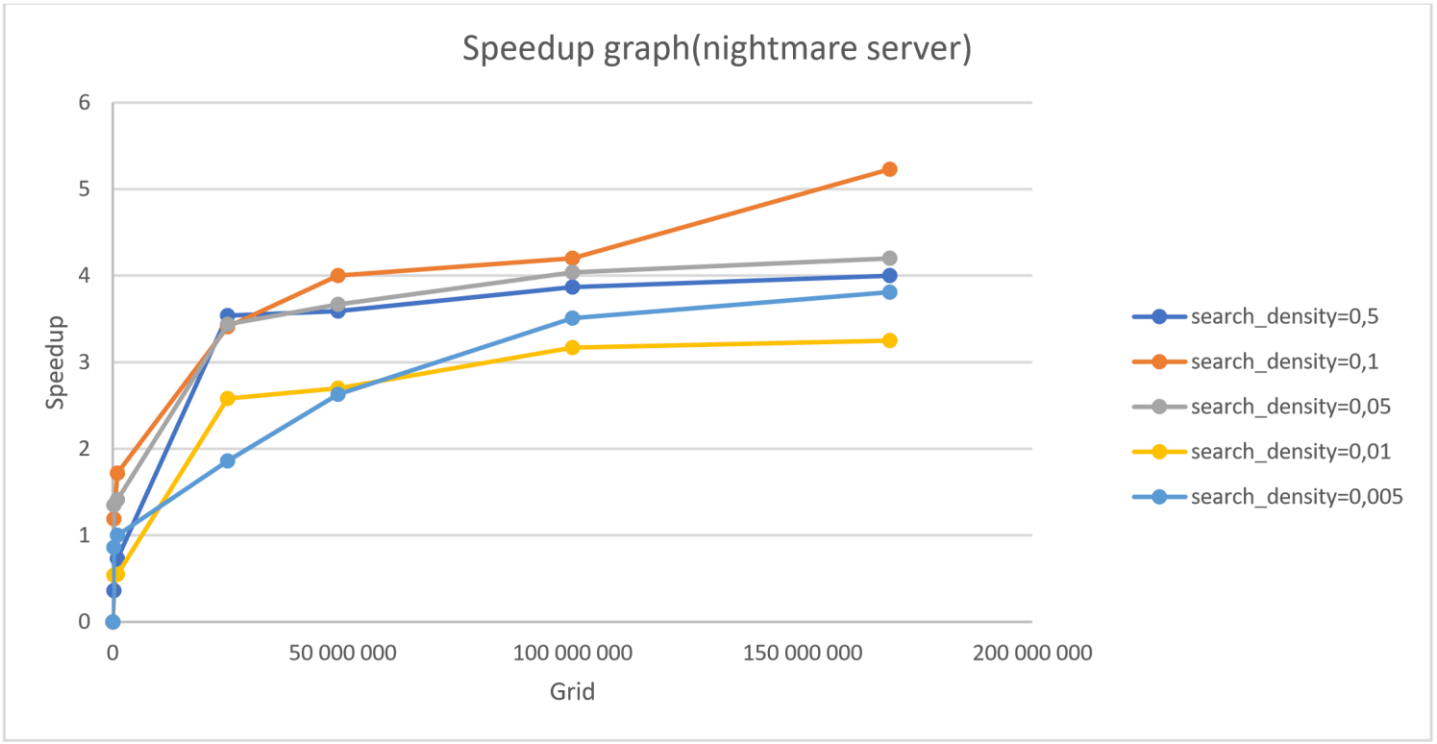
By using ForkJoin framework, the code uses parallel execution and utilises multiple CPU cores, speeding up the computation. Implementing a cutoff prevents parallel overhead for small tasks and task splitting helps distribute the workload among threads. Each subtask independently searches for the minimum value, reducing contention for shared resources.

Validation.

I was Comparing outputs(minimum) from the given serial program against my parallel program(using Rosenbrock function) and they were giving same output. I have tested my parallel program with boundary cases, small and large input sizes, it responded accordingly (fast for large input size). Debugging the code to track the behaviour by checking initial values, state during the computation and the final state of variables. Monitoring processor utilisation across all cores(4 and 8), for the serial program, I noticed that there is 100% utilization on one CPU at a time but for serial all CPU's were 100% during execution.

Benchmark


Speedup graph(Laptop)
Speedup
Grid size
0
1
2
3
4
5
6
0
50 000 000
100 000 000
150 000 000
200 000 000
search_density=0,5
search_density=0,1
search_density=0,05
search_density=0,01
search_density=0,005


Speedup graph(nightmare server)
Speedup
Grid
0
1
2
3
4
5
6
0
50 000 000
100 000 000
150 000 000
200 000 000
search_density=0,5
search_density=0,1
search_density=0,05
search_density=0,01
search_density=0,005

Above are speedup graphs which was my main experiment. I used my laptop and nightmare server as they have different hardware specifications to observe the impact on performance. I used same programs and arguments for the program to ensure fairness and consistency. I was running the program manually for each arguments (change grid size and search density) and recording everything. I used excel to analyse my data and create speedup graphs.

Machine Architectures

Laptop: 8gb ram, 4 cores(8 logical processors), 64-bit operating system, x64-based processor, @1,60GHz

Nightmare server: 48gb ram, 8 cores, Architecture: x86_64, CPU op-mode(s): 32-bit, 64-bit, @2,40GHz

problems/difficulties

My laptop had the error “java.lang.OutOfMemoryError: Java heap space” due to my low ram(memory) when trying to compute high grid values with high search density, but it worked on server which has high ram(memory). I have also noticed that my program is slightly slow on server even though the server had high specifications, this may be due to a lot of other students running their programs concurrently causing CPU to be heavily utilised. Doing traditional experiments is tedious, I may have not covered all possible paths and due to OS loads(other programs running on the OS) and warm up effects, I was getting different times always, causing me to pick median.

Results

Even though the two machines produced different outputs(time), the parallel program demonstrates a good performance across a range of grid sizes (particularly in 5000*5000 to 13000*13000 grid sizes) for both machines. As the grid size increases, the program’s speed was showing good results, and this is because larger grid sizes provide more opportunities for parallelization. Decrease in search density which decreases number of searches showed improvement on both serial and parallel speed. I did not cater for SEQUENTIAL_CUTOFF experiment as it had little effect on the speed compared to grid size and search density, so my SEQUENTIAL_CUTOFF is 5000 (optimal). SEQUENTIAL_CUTOFF is for managing small sample size, and both programs were performing equally for small sample size thus it is inefficient for larger sizes which is what I was my interest.

The maximum speedup achieved was 5,23 by the server. While this is an improvement over the sequential program, it is not meeting the ideal speed of 8 on 8 cores. This may be due to Amdahl’s law(serial execution having high percentage). The measurements are reliable because I was making multiple runs to account for variations, warm-up before actual measurements began and consistent environment (same machines and same arguments on both machines).

Conclusion

Overall speedup experiment provides valuable insights into the behaviour of the parallel program and the impact of different parameters on its performance. It was worth it to parallelise the program as it

shows speed improvements for a wide range of grid sizes and search densities. While there were inconsistent outputs(time) for larger problem size, the parallel version consistently outperformed serial program and I have utilised available computational resources.