Name: Dylan Friedman
Student Num: FRDDYL002
Course: CSC2002S
Date: 09 August 2023

Serial Solution vs Parallel Solution

Methods description:

The MonteCarlo serial solution was provided with a Search Class to provide the next direction in regards to the height of a point on the graph that is being computed. The solution I used was to create a similar MonteCarloParallelMinimization Class that implement a Parallel Class ([1] extending the RecursiveTask base) into the Fork/Join framework, this computes the searches in parallel and then compare the forked off threads to thus provide a quicker search for the Global minimum as well as the x and y values of that point.

[1] The RecursiveTask Class uses an inner class, Result Class (in Parallel Class), as a generic data type. The Result Class stores the global minimum as "min" and the index of which search "min" is found.

Optimising: To optimise data capture, in the MakeFile [2] serial and parallel runs 5 times so such that the median of the data can easily be captured. I kept the Sequential Threshold as a constant which is the total number of searches divided by the number of CPUs used on my Pc, keeping in mind that the different sequential cut offs do not vary the SpeedUp to a significant extent.

[2] "make serial" & "make parallel" in the command line

A

```
dylan@DylanFriedman:/mnt/c/Users/dylan/OneDrive/Desktop/MonteCarloMin/bin$ java.exe MonteCarloMini.MonteCarloMinimization 100 100 1 100 1 100 0.5
Run parameters
        Rows: 100, Columns: 100
        x: [1.000000, 100.000000], y: [1.000000, 100.000000]
        Search density: 0.500000 (5000 searches)
Time: 8 ms
Grid points visited: 4160  (42%)
Grid points evaluated: 22428  (224%)
Global minimum: 0 at x=1.0 y=1.0
```

B

```
dylan@DylanFriedman:/mnt/c/Users/dylan/OneDrive/Desktop/MonteCarloMin/bin$ java.exe MonteCarloMini.MonteCarloParallelMinimization 100 100 1 100 1 100 0.5
Run parameters
        Rows: 100, Columns: 100
        x: [1.000000, 100.000000], y: [1.000000, 100.000000]
        Search density: 0.500000 (5000 searches)
Time: 11 ms
Grid points visited: 3866  (39%)
Grid points evaluated: 14542  (145%)
Global minimum: 0 at x=1.0 y=1.0
```

Validation: The Rosenbrock function returns the same inputs running the Serial & Parallel solutions.

Benchmarking: The program was run on MyPc and the SeniorLab, 8 CPU and 4 CPU respectively. The variables that I kept constant was the Cutoff Threshold and the Search Density - Search Density varied between 0.5 and 1 as stated in the results section. The data ranged in Rows/Columns and x & y values. Refer to the **Discussion** in the **Results Section.** The data size ranged from 100 then 250 to 5000 in increments of 250. With each data point, the code was run 5 times to then get the median. Thus 20 data points * 5 runs * 2 different constants * 2 different systems = 400 results were obtained with 80 results (the median) being used.

Architecture: My Pc - Dell Inspiron 15 3511

```
dylan@DylanFriedman:/mnt/c/Users/dylan/OneDrive/Desktop/MonteCarloMin/bin$
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
  On-line CPU(s) list:   0-7
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
    CPU family:          6
    Model:               140
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            1
    BogoMIPS:            4838.39
```

Architecture: SeniorLab

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz
    CPU family:          6
    Model:               158
    Thread(s) per core:  1
    Core(s) per socket:  4
    Socket(s):           1
    Stepping:            11
    CPU(s) scaling MHz:  22%
    CPU max MHz:         3600.0000
    CPU min MHz:         800.0000
    BogoMIPS:            7200.00
```

Problems/Difficulties: The data captured clearly shows the trend of the Speed Up Graphs. Although the data captured was with a Parallel solution that did have a key error of not comparing the Threads properly, this error was corrected but the results from the unfixed code were used. There were difficulties in the automation of the collection of data, and thus manually collecting the data was done. The implementation of the Parallel Class was straightforward.

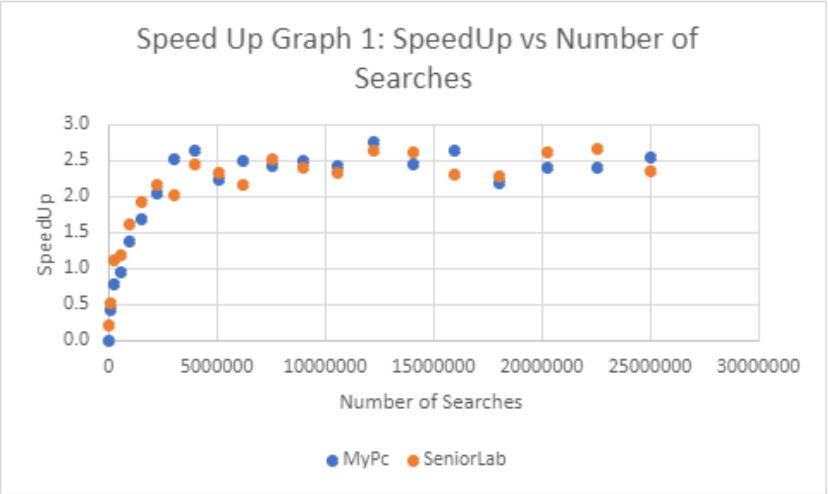
The previous code worked but did have inconsistent validation as shown:

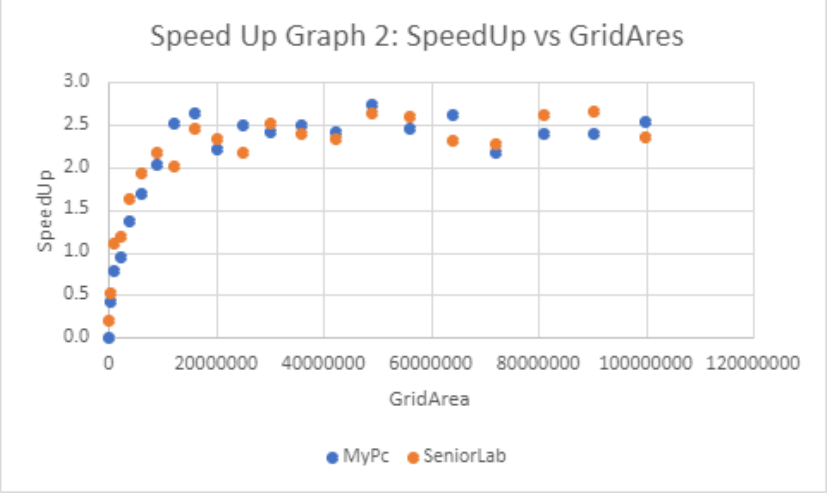
```
dylan@DylanFriedman:/mnt/c/Users/dylan/OneDrive/Desktop/MonteCarloMin/bin$ java.exe MonteCarloMini.MonteCarloParallelMinimization 1000 1000 -100 100 -100 100 1
Run parameters
        Rows: 1000, Columns: 1000
        x: [-100.000000, 100.000000], y: [-100.000000, 100.000000]
        Search density: 1.000000 (1000000 searches)
Time: 106 ms
Grid points visited: 345309  (35%)
Grid points evaluated: 1133273  (113%)
Global minimum: 2000 at x=1.2 y=1.4

dylan@DylanFriedman:/mnt/c/Users/dylan/OneDrive/Desktop/MonteCarloMin/bin$ java.exe MonteCarloMini.MonteCarloParallelMinimization 1000 1000 -100 100 -100 100 1
Run parameters
        Rows: 1000, Columns: 1000
        x: [-100.000000, 100.000000], y: [-100.000000, 100.000000]
        Search density: 1.000000 (1000000 searches)
Time: 102 ms
Grid points visited: 377975  (38%)
Grid points evaluated: 1292016  (129%)
Global minimum: 0 at x=1.0 y=1.0
```

Note: The new code is consistently valid.

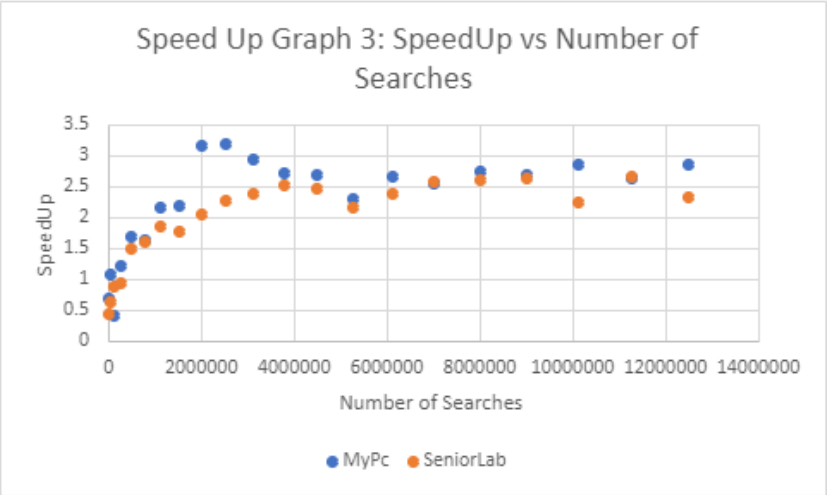
Results Section:

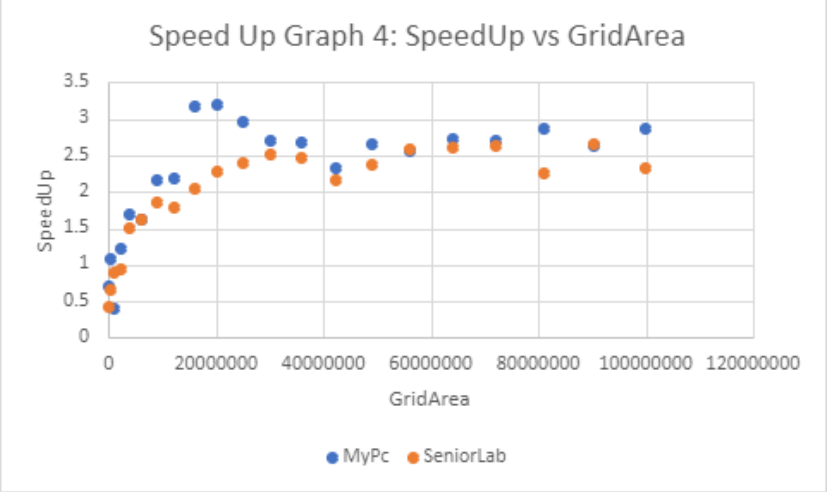
Graphs 1 & 2 have Search Density of 1



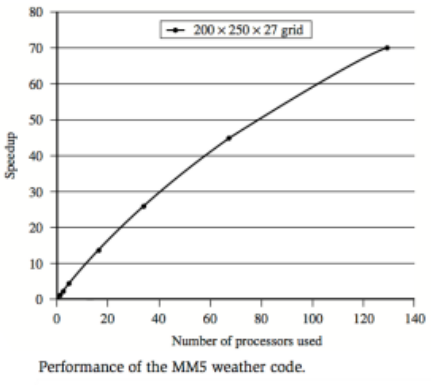


Graphs 3 & 4 have Search Density of 0.5





Discussion: The data obtained shows the trend of the search density as the search parameters increase. As seen from the graph a logarithmic trend can be made out, showing that the Parallel solution worked effectively, as compared to this graph here:



Performance of the MM5 weather code.

The range of grid sizes varied from 100 to 5000 and increased in increments of 250 from 250. The GridArea was calculated as (xMax-xMin)*(yMax-yMin) and compared to SpeedUp in **Graphs 2 & 4**. The Number of searches varied from 0 to 25 million with 100 to 5000 as input and increased in increments of 250 from 250. The number of searches was calculated as (rows*columns*search density)

Analysis: The best performing range with a SpeedUp of 3.16 and 3.19 (which is close to the ideal SpeedUp of 4), this is from 20 million to 25 million searches respectively, then plateauing at 60 million searches. The best performing grid area is of 16 million and 25 million, then plateauing at 50 million grid area.

The reliability of the data has been negatively impacted by the fact that the code had been altered after collecting the data.

Anomalies: The smaller grid size gave way to Serial being more efficient than Parallelisation, this is due to Parallelisation having to delegate and organise tasks before computing and the plateauing of the SpeedUp is due to the maximum amount of cores being used at their maximum clock speed.

Conclusion: The parallelization of the solution proved useful for bigger grid sizes and a higher number of searches. Before the level (10M grid area & 10M searches =1250 column/rows/xmax/ymax & -1250 xmin/ymin) the serial solution performed better. I.e. For higher order computations or computing a large amount of data Parallelisation is very optimal.

The graphs show that with 0.5 search density the SpeedUp is higher than a 1 search density, which validates that the parallel solution performs well with a large amount of data to compute.