CSC2002S Assignment PCP1 2023
Parallel Programming with Java:
Parallelizing Monte Carlo Function Optimisation

set by M. Kuttel, adapted from "Hill Climbing with Montecarlo"
EduHPC'22 Peachy Assignment

1. Introduction

This assignment will give you experience in programming a parallel algorithm for the shared memory parallel programming in Java. You will validate your parallel solution for correctness and benchmark it against the serial algorithm to determine under which conditions parallelization is worth the extra effort involved.

This assignment will also expose you to a nice example of a Monte Carlo method. Monte Carlo methods use random selections in calculations to solve numerical or physical problems. In this assignment we deal with a Monte Carlo algorithm for finding the **minimum** (the lowest point) of a two-dimensional mathematical function *f(x,y)* within a specified range – this is an optimization problem. You can think of the 2D function as representing the height of a terrain, and the program's task is to find the lowest point in a specified rectangular area.

This area is represented as **discrete grid** of evenly spaced points. As this grid may be huge, and the cost of computing the function for all the points is high, the Monte Carlo algorithm instead employs a probabilistic approach to finding a minimum of the function without computing all the values in the grid. This works with a **series of searches**, as follows.

For each search, a starting grid location is chosen **randomly** and its height is calculated. The search then attempts to move **downhill** from that point, by calculating the height of all **four neighbouring grid points** and then moving to the one which has the lowest value. From this new grid point, it then attempts to move downhill again, in the same way. The search continues until it can't find a downhill direction: all the neighbouring points have a higher value than the current point. At this point the search has found a local minimum and stops.

With enough separate searches, this algorithm can find a "global" minimum (the minimum point in the region of interest) with a high probability. If you run more searches, there is a greater the chance of finding the global minimum, but also a concomitant increase in computational cost.

2. The serial solution

You are provided with a Java program that codes a serial solution to the problem. The program hard codes the function that we will use for this assignment, but is generally applicable to other suitable functions. Your first step should be to have a good look at the code.

The code has a few optimizations, as follows.

- In order to avoid computing the costly function for all the grid points in advance, the grid is initialised to large values. The function value is then only calculated at grid point if/when a search checks that point.
- When a search moves to a point already visited by a previous search, it stops – as it would follow then exactly the same path to the same local minimum.

After all searches have completed, the program outputs:

- all the run parameters given as command-line arguments;
- the total time taken;
- the number of grid points visited/evaluated
- and the **result**: the location and value of the global minimum found.

2.1 Program arguments

The program takes the following **command line arguments** in order:

- rows, columns: the number of rows and columns in the discrete grid representing the function;
- xmin, xmax, ymin, ymax: the boundaries of the rectangular area for the terrain;
- searches density: number of searches per grid point;

2.2 Java classes

There are three main classes used in this the fairly simple program, as follows. The `TerrainArea` class represents the terrain/function; the `Search` class represents a search of a `TerrainArea` object; and the `MonteCarloMinimization` class main class intialises everything and collates the final result.

2.3 Debug mode

If the DEBUG flag is set to true, the program will output text representations of the grid heights evaluated and the searches' paths on the grid.

Once you have had a look at the code, run it to see how it works. Experiment with different ranges for the function, and different numbers of rows and columns, as well as different search densities.

3. Your parallel solution

Your task in this assignment is to create a fast parallelization of the serial algorithm. As each search is independent of the others, this algorithm is almost embarrassingly parallel, and so should be straighforward to parallelise.

Note that parallel programs need to be both correct and faster than the serial versions. Therefore, you need to demonstrate both correctness and speedup for your assignment. If speedup is not achieved, you need to explain why.

3.1 Specific requirements

In this assignment, you must do the following.

1. Profile the **serial program,** measuring the time taken to run for a range of grid sizes and searcher densities.
2. Write **parallel version** using the Java Fork/Join framework in order to speed up the algorithm.
3. Validate your parallel version to demonstrate that it works correctly (i.e. produces the same solution as the serial version). The Rosenbrock function -

$$f(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2$$

   - could be useful for this process, as is often used as a test problem for optimization algorithms.
4. **Benchmark** your parallel program experimentally, **timing** the **execution** on at least two machines (e.g. your laptop and the departmental server) with **different grid sizes** and **search densities**, generating **speedup graphs** that show the conditions under which you get the best parallel **speedup**. (Do not give timing graphs)
5. Write a **short report** including the graphs with an explanation of your findings.

3.2 Race condition

Note that your parallel solutions will have to share a `TerrainArea` object. This will create a race condition in a case where threads write simultaneously to the same grid location. However, in this case for this assignment **we will ignore the race condition**. This is justifiable, as the race condition is fairly benign, and should just result in an occasional redundant evaluation of the function and will not invalidate the final result. Protecting against the race condition will most likely cost more in computational time than the extra function evaluations. The cost of protecting against this in terms of time is likely to be more than the benefit gained.

4. Report

You must submit a short assignment report **in pdf format** (please do not submit Word documents). Your **concise** report must contain the following:

A **brief** *Methods* section describing:

- your **parallelization approach,** including any particular issues/considerations and **optimisations** you implemented;
- how you **validated** your algorithms (showed that it is correctly implemented);
- how you **benchmarked** your algorithm,;
- the **machine architectures** you tested on; and
- any **problems/difficulties** you encountered.

A *Results* section, with **speedup graphs**.

- Graphs must show how the parallel algorithm scales with **grid size** and number of **searches**, and on (at least 2) different computers, one of which needs to be a

mulitcore machine like the departmental server. **Graphs** should be clear and **labelled** (title and axes).

- **A brief discussion** that answers following questions must be included. This should answer the following questions.
    - For what range of grid sizes does your parallel program perform well?
    - What is the maximum speedup you obtained and how close is this speedup to the ideal expected?
    - How reliable are your measurements?
    - Are they any anomalies and can you explain why they occur?

A *Conclusions* (note the plural) section where you say whether it is worth using parallelization (multithreading) to tackle this problem in Java.

Please do NOT ask for the recommended numbers of pages for this report. It should be short, with clear graphs. Say what you need to say: no more, no less.

5. Assignment submission requirements

You will need to submit an **assignment archive**, named with your **student number** and the **assignment number e.g. KTTMIC004_CSC2002S_PCP1.** Your submission archive must contain

- a **short report (in pdf format)**
- **your parallel program** comprising three Java program files containing your solution, **named** `TerrainArea.java`; `SearchParallel.java`; and `MonteCarloMinimizationParallel.java`. Use these **exact names**.
- a **Makefile** for compilation
- A readme file explaining how to run the program (you cannot change the input parameters)
- **a GIT usage log (**as a .txt file, use `git log –all` to display all commits and save).

Upload the archive file and **then check that it is uploaded.** It is your responsibility to check that the uploaded file is correct, as mistakes cannot be corrected after the due date.

The deadline for marking **queries** on your assignment is **one week after the return of your mark.** After this time, you may not query your mark.

1.6 Assignment marking

A draft marking guide is included below.

Marking Sheet: CSC2002S PCP 2023 Parallel Assignment	***Maximum Mark***
CODE	
Readme file	*1*
Good clear code style and organization. Efficient algorithms with no odd/unnecessary code.	*2*
Code comments are clear and informative and placed where needed.	*2*
REPORT: Introduction and methods	

Parallelisation approach clearly explained in sufficient detail.	*2*
Validation clearly described.	*2*
Benchmarking correctly performed and clearly explained - at least 2 machine architectures, good range of data sizes and search densities. Sufficient detail provided so that someone else could do the same thing.	*2*
REPORT: DISCUSSION	
Speedup Graphs: Graphs of parallel speedup versus grid size on two different architectures and with varying search denisities . NOTE THAT THESE MUST BE SPEEDUP GRAPHS!	*2*
Graph presentation/format/design: clear and easy to understand/comprehend. All axes labelled.	*1*
Discussion of the range of data sizes for which the best speedup is obtained, and why, well justified.	*3*
Discussion of how the maximum speedup compares to the ideal for each architectur	*2*
Trends and any anomalies (spikes etc.) in the graphs discussed and explained.	*2*
CONCLUSIONS	
Was parallelization worth it? - conclusions justified and explained. The conclusions make sense in the context of the results presented in the report.	*2*
Clarity of the discussion - how clear and understandable is the report, especially the presentation and discussion of the results?	*2*
TOTAL	***25***
Code does not execute/run.	**-5**
no GIT repository.	**0**
GIT repository, but no regular updates (commits on at least on 5 separate days).	**0**
No makefile.	**-2**
Late penalty (10% for first day or part thereof). No late handins after one day late!	***-2.5***

Note: submitted code that does not run or does not pass standard test cases will result in a mark of zero. **Any plagiarism, academic dishonesty, or falsifying of results reported will get a mark of 0 and be submitted to the university court**.