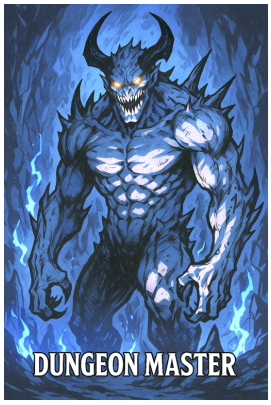
CSC2002S
PCP1 Assignment 2025
Parallel Programming with Java:
Hunt for the Dungeon Master:
A Parallel Search Inspired by Solo Leveling

version 1, set by M. M. Kuttel[1,2]



1. Introduction

This first PCP assignment will give you experience in programming a parallel algorithm for the shared memory parallel programming model with 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 takes as inspiration the world of *Solo Leveling*, where dungeons appear unpredictably, guarded by powerful dungeon masters. Your task in this assignment is to implement a parallel search algorithm to enable the Shadow Monarch, Sung Jin-Woo, to use his Shadow Army to locate the dungeon master, hidden somewhere in a two-dimensional dungeon grid. The dungeon master is detected by the level of power, or "mana", that emanates from it: the point in the dungeon of highest mana is the location of the dungeon master.

The search implemented in this assignment is a nice example of a Monte Carlo method. Monte Carlo methods use random selections in calculations to solve numerical or physical problems. Here we use a Monte Carlo algorithm to find the **maximum** (the highest value) of the "mana" within a dungeon, which is calculated as a two-dimensional mathematical function *f(x,y)*– this is an **optimization problem**. This dungeon is represented as a **discrete grid** of evenly spaced cells. 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 mana value is calculated. The search then attempts to move '**uphill'** from that point, by calculating the mana of all **eight neighbouring cells** and then moving to the one which has highest value. From this new grid point, it then attempts to move uphill again, in the same way. The search continues until all the neighbouring points have a lower mana value than the current point. At this point the search has found a local maximum and stops. With enough separate searches, this algorithm can find a global maximum with a high probability. If you run more searches, there is a greater chance of finding the global maximum, but also a concomitant increase in computational cost.

Your task in this assignment is to code a parallel solution to this problem, validate it against the serial solution for correctness and then benchmark it against the serial algorithm to determine under which conditions (if any) the parallelization was worth the extra programming effort required.

---

[1] Adapted from "*Hill Climbing with Montecarlo*", Peachy Parallel Assignments (EduPar 2022), 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

[2] Set with some help from ChatGPT in reframing and generating the complex mana function; theme suggested by CSC3002F 2025 students.

[3]Solo Leveling is an anime television series adaptation produced by A-1 Pictures based on a South Korean fantasy web novel written by Chugong (recommended!).

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. **You may not change this function.** 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 low 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 then follow exactly the same path to the same local minimum.

Your first step should be to have a good look at the code and figure out how it works. Then run it, experiment with different inputs with differing numbers of rows and columns and cell values. Examine the output.

3. Your parallel solution

Your task in this assignment is to create a parallel version of the serial algorithm that is correct and runs faster and uses the threading library discussed in class. You can only use the **join()** synchronisation mechanism in the assignment – no other synchronisation mechanisms are allowed (e.g. no barriers, synchronised methods, no atomic variables etc.) You may use generative AI (e.g. ChatGPT) to assist with this, but you will need to acknowledge how it was used and adhere to the requirements of the assignment.

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 over the serial solution is not achieved, you need to explain why this happens.

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 input sizes and starting values for the cells.
2. Write **parallel version** using the framework discussed in class in order to speed up the algorithm (no other parallel approach is allowed). This version **must have the same input, output text and file output as the serial solution** (although you can output other files as well, if you choose).
3. **Validate** your parallel version to demonstrate that it works correctly (i.e. produces the same solution as the serial version).
4. **Benchmark** your parallel program experimentally, **timing** the **execution** on both your **laptop** and the **departmental server** with **different input sizes**, generating **speedup graphs** (not raw timing graphs) that show the conditions under which you get the best parallel **speedup.**
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 map 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 benign; at worst, it may cause a few redundant function evaluations but will not change the correctness of the final result. Protecting against the race condition will most likely cost more in computational time than the extra function evaluations.

4. Report

You must submit a short assignment report (no more than 5 pages) **in pdf format** (may not submit Word documents). Your **concise** report must contain the following:

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

1. **Validation:** how you **validated** your parallel algorithm (showed that it is correctly implemented). This section is extremely important.
2. **Optimum search density:** how you did you determine an optimum search density for benchmarking?
3. **Benchmarking:** explain clearly how you **benchmarked** your algorithm, in enough detail that someone could reproduce your work, including the **machine architectures** you tested on, the range of inputs tested and how you determined a **good sequential cutoff**.

A *Results* section containing the following.

1. **Validation:** evidence that your parallel implementation is correct.
2. **Benchmarking: speedup graphs** demonstrating how the parallel algorithm scales with **grid size** and number of **searches** on your laptop and the departmental server. **Graphs** should be clear and **labelled** (title and axes).
3. **A brief discussion** that answers following questions: For what input 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 there any anomalies and can you explain why they occur?
4. **Conclusions:** Is it worth using parallelization (multithreading) to tackle this problem in Java?

The report must be no more than five pages, including graphs. To do this, have multiple graphs on a single axis.

5. Assignment submission requirements

You will need to **submit** your assignment in two places:

1. Submit the **code** to the **automarker** as a zipped **archive containing:**

- **all the files** needed to run your solution.
- your parallel program comprising three Java program files **DungeonHunterParallel.java**; **DungeonMapParallel.java**; and **HuntParallel.java**. Use these **exact names,** do not add additional files into your parallel solution.
- a **Makefile** for compilation **that works as the serial solution does**, e.g. the following command must run your parallel solution correctly.

```
make run make run ARGS="100 0.2 0"
```

- **a GIT usage log (**as a .txt file, use `git log –all` to display all commits and save).

2. Submit your report (**pdf format only**) to the **Amathuba assignment**.

*NOTE: if you do not follow these instructions exactly, or submit an incorrect assignment, or submit it to the incorrect place, your **mark will be zero**. The onus is on you to check and double-check your final submission.*

If you try to submit one minute after the final cut-off, this counts as no submission. Do not email about this – you need to manage your time and submit on time.

*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.*