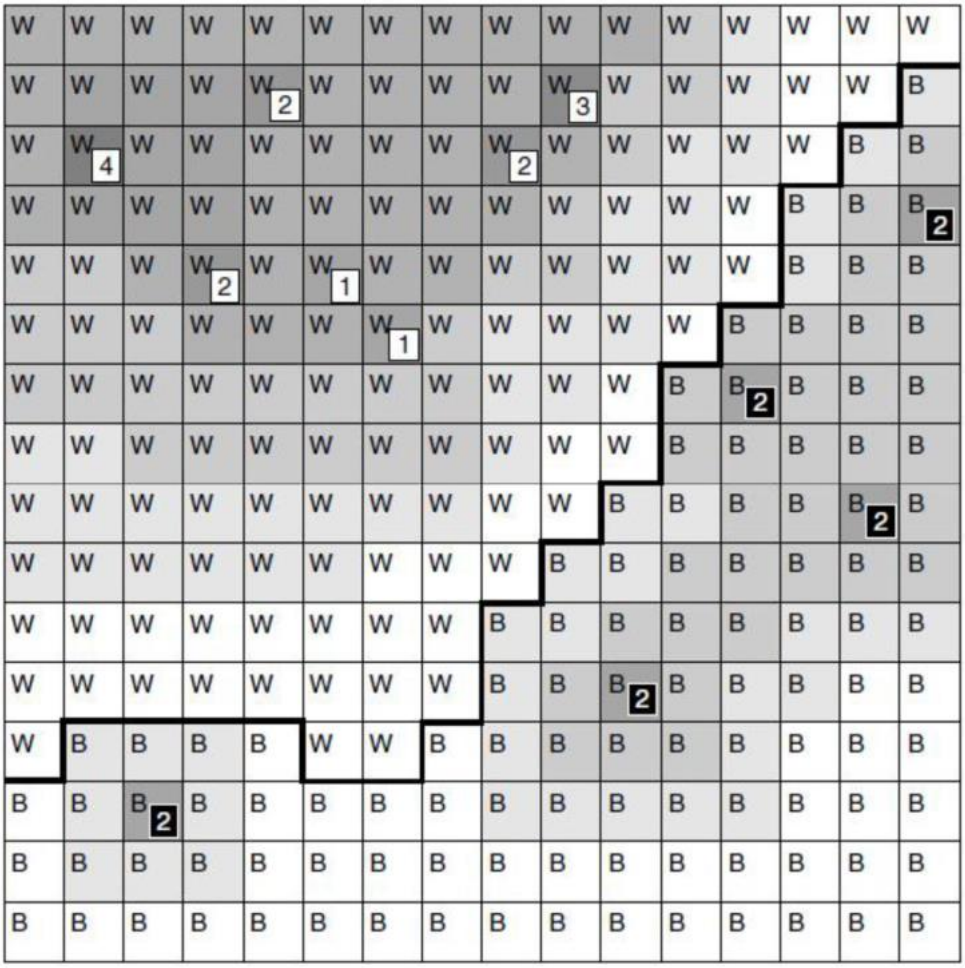
MiniProject 1 (Parallel computation)

Background

One war strategy game has a field of 300 x 300 squares. On the squares are placed the military units of the parties, whose strength is represented by numbers. Positive values indicate AI units, and negative values represent the player's units. The higher the absolute value of the number, the stronger the unit.

The game's AI uses a so-called "**influence map**" to support its decision-making, which also consists of 300 x 300 squares. Each square in the influence map is used to sum up the total impact of the entities of both parties. Below is an example of a small influence map (the squares are coloured according to which party's impact is greater):



The locations and strengths of the AI units are marked in white above, the player's in black. Impact map is obtained by adding together the strengths of all units in the game, but with the further away a unit is, the smaller its effect:

$$I_d = \frac{I_0}{1 + d}$$

I_d is the influence of the unit on some other square, when the unit strength is I_0, and distance from the unit to the square is d.

In main.cpp you can find the definition of the "map" and the corresponding impact map ("influence_map"). Both are 2-dimensional tables. The auxiliary function distance() calculates the distance between the two coordinate points.

Small Project 1 (Parallel Computation 10p)

The task is to write an algorithm to calculate the impact map based on the game map. The algorithm to be created is parallelized in different ways. If you want to get fancy, you can also implement a graphical visualization of the generated impact map (not mandatory).

Part 1 (Serial algorithm)

Write an algorithm to calculate the impact map based on the game map. First, place the map of units of both sides (remember to reset remaining empty squares).

The calculation algorithm should work as follows:

- The outer loops go through each coordinate (c0, r0)
  - Initialize the impact map point [c0][r0] to 0
  - Inner loops pass through each coordinate (c1, r1)
    - I_0 is the value of the map point [c1][r1]
    - d is the distance between coordinates (c0, r0) and (c1, r1)
    - Add the effect of this game map point [c1][r1] to the map point [c0][r0] of the impact map (use I_0 and d calculated by you in the impact formula)

Summa summarum: go through each square of the map. Sum the effect of all the other squares in the map (the effect decreases the further away the unit is).

Implement the algorithm using the "brute force" method described above (there are perhaps much smarter ways to do this exist...). Take a look at the std::chrono class and use it to measure how long the serial algorithm to execute an algorithm. If it takes an insanely long time, you can reduce the size of the map. Also release compilation mode can also help.

Check how many cores your machine has. What percentage of your machine's potential of your machine's potential computing power is used by the algorithm you are writing (see TaskManager, etc.).

Part 2 (Parallelization using asynchronous function calls)

Parallelize the serial algorithm using asynchronous function calls (std::async). Remember to wait for all asynchronous function calls to finish (std::future).

Measure the execution time.

In parallel versions, it may not make sense to parallelize the computation of the impact map with influence map frame at a time - try to find ways to give a larger overall to compute for a single asynchronous function call or thread.

Part 3 (Parallelization using threads and mutual exclusion)

Parallelize a serial algorithm by creating threads (std::thread). Be sure to wait for all threads to complete (join).

Add a global counter to the implementation

```
int ready_count = 0;
```

Your algorithms should add a counter every time one cell of the influence map is completed.

Check the counter value when the count is finished. Is the value COLS * ROWS? If not, why not? Correct the situation with a lock (std::mutex).

Measure the execution time both for a version with locking enabled and for a version with lock is not used. What could be the reason for the difference?

Part 4 (Parallelization using standard library algorithms)

Check out the standard library (STL) function std::for_each, which allows you to specify an operation for each element of the collection. The operation can be defined as a lambda function or as a normal function.

Modify the Part 1 algorithm to use for_each. Make sure that it still works correctly.

Then define a parallel execution for the for_each algorithm std::execution::par_unseq (this requires at least C++17 support from the compiler). How much did the execution time drop?

Part 5 (Better serial algorithm)

In most cases, the best solution to solve the performance problem is not parallel computing, but using a better serial algorithm.

So, **design and implement a more efficient serial algorithm**, assuming that both parties have, after all, a rather finite number of units on the map. Finally, measure the execution time.