Exercises 8

Task 8.1. OpenMP Quicksort

Use OpenMP to speedup a Quicksort array sorting procedure as follows.

First, implement a standard Quicksort algorithm. You can use Lomuto partition scheme as the easiest to code:

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Next, note the fragment

```
quicksort(A, lo, p - 1)
quicksort(A, p + 1, hi)
```

Here you can use independent OpenMP sections to run these calls in parallel. However, a straightforward implementation produces excessive thread creation calls due to recursion and causes significant performance degradation. The easiest way to fight it is to parallelize sorting of large pieces of the initial array only.

For example, you can try running `quicksort()` calls in parallel if they both process at least `N` / 30 elements (where `N` is the source array size), keeping the calls sequential otherwise.

Benchmark your solution and report how much speedup is possible to obtain with OpenMP comparing to a non-parallel solution.

Task 8.2. Crack another Password v2

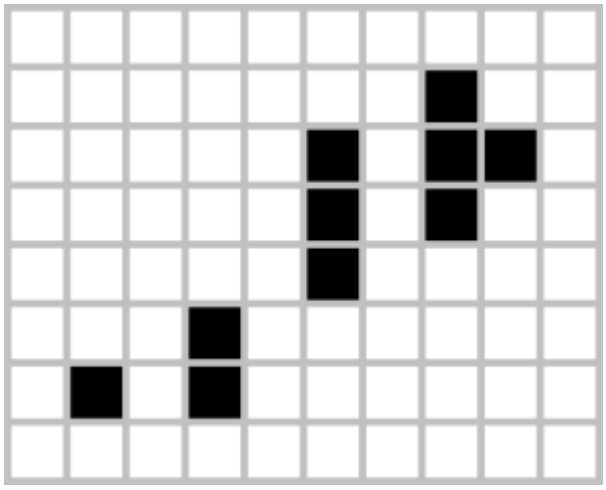
Use OpenMP to solve the problem `2.3 Crack the Password v2`. If your program uses nested for-loops, you can simply parallelize the outer loop and keep the rest of the code unchanged. Once a password is found, print it, and exit the program.

Task 8.3. Find More Primes

Use OpenMP to write a program that outputs all the prime numbers in the user-specified interval [A, B]. The numbers can be printed in any order.

Task 8.4. OpenMP Life

"Game of Life" is a *cellular automaton* invented by J. Conway in 1970. The point of this simulation is to observe the development of a user-defined colony of *cells*, "living" on an infinite 2D board:



–

To compute the next state of the game board we add or remove cells according to the following rules:

1. A cell having 2 or 3 neighboring cells, stays intact, any other cells die.
2. A new cell is born on an empty square if it has exactly 3 neighboring cells.

Each square is considered to have 8 neighbors (horizontally, vertically, or diagonally adjacent). Note that all changes are applied *to the next state*, so in practice it means that we generate a new game board every time, rather than modify the existing board.

Your task is to simulate N steps of "Game of Life" using OpenMP. The most straightforward solution would be simply to parallelize each loop in the main data processing code. Print out the state of the game board after each step.

Use the following hints and suggestions:

1. You can read game configuration from a text file or simply hardcode it in your solution.
2. Simulate a 6×6 game board. Treat leftmost/rightmost columns and top/bottom rows as adjacent (so the board will have a donut-shaped *toroidal* form).
3. Keep game state in a simple 2D Java array. To calculate the next board state, create another array and write to it concurrently from all your threads.

Test your program on the following classic configurations:

Blinker (should switch between vertical and horizontal positions):

Block (should stay intact):

Glider (should “glide” across the game board):