Epidemics Simulation

With OMP threads

Blejdea Emil Alexandru 3C 1.1

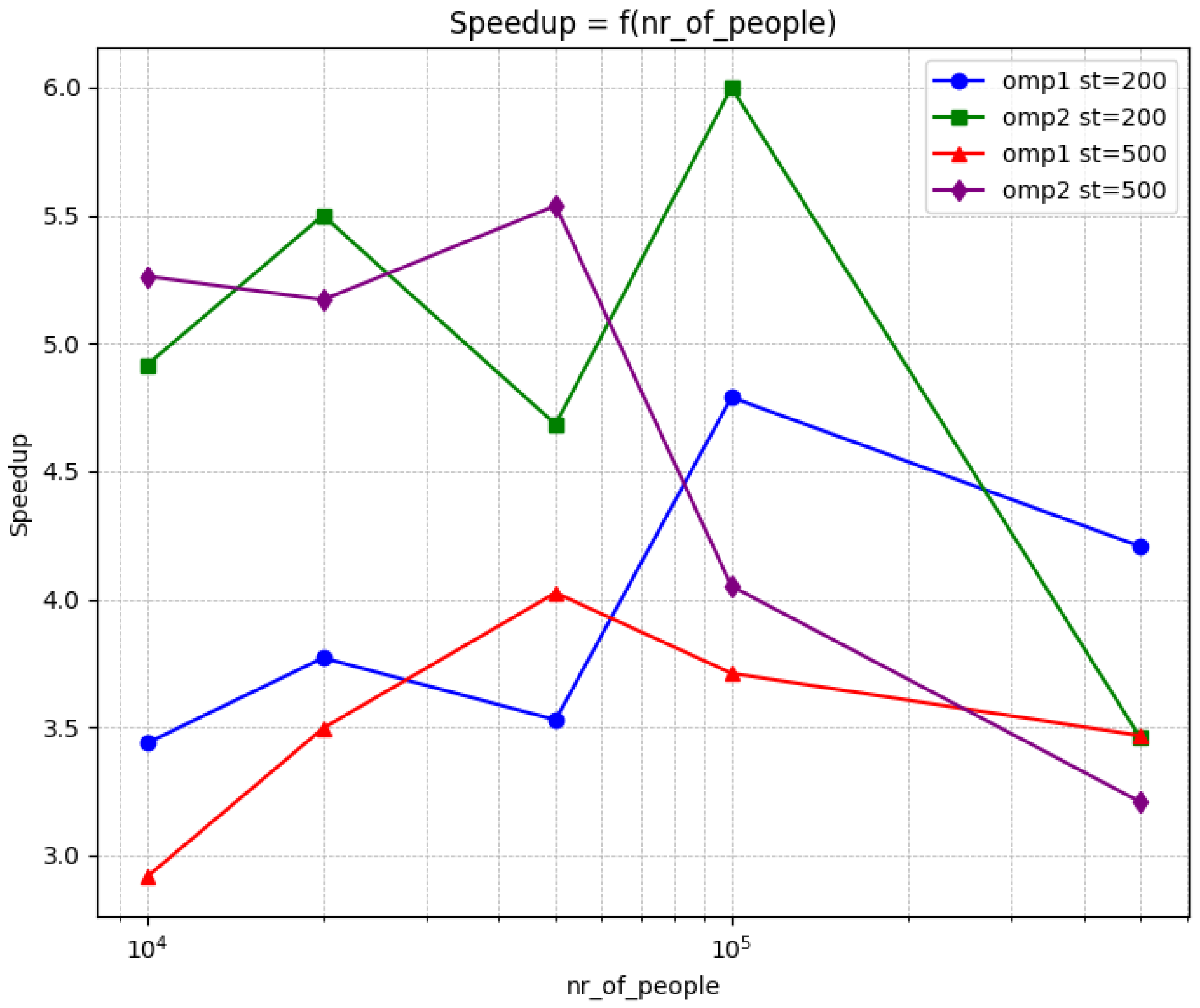
Functionality

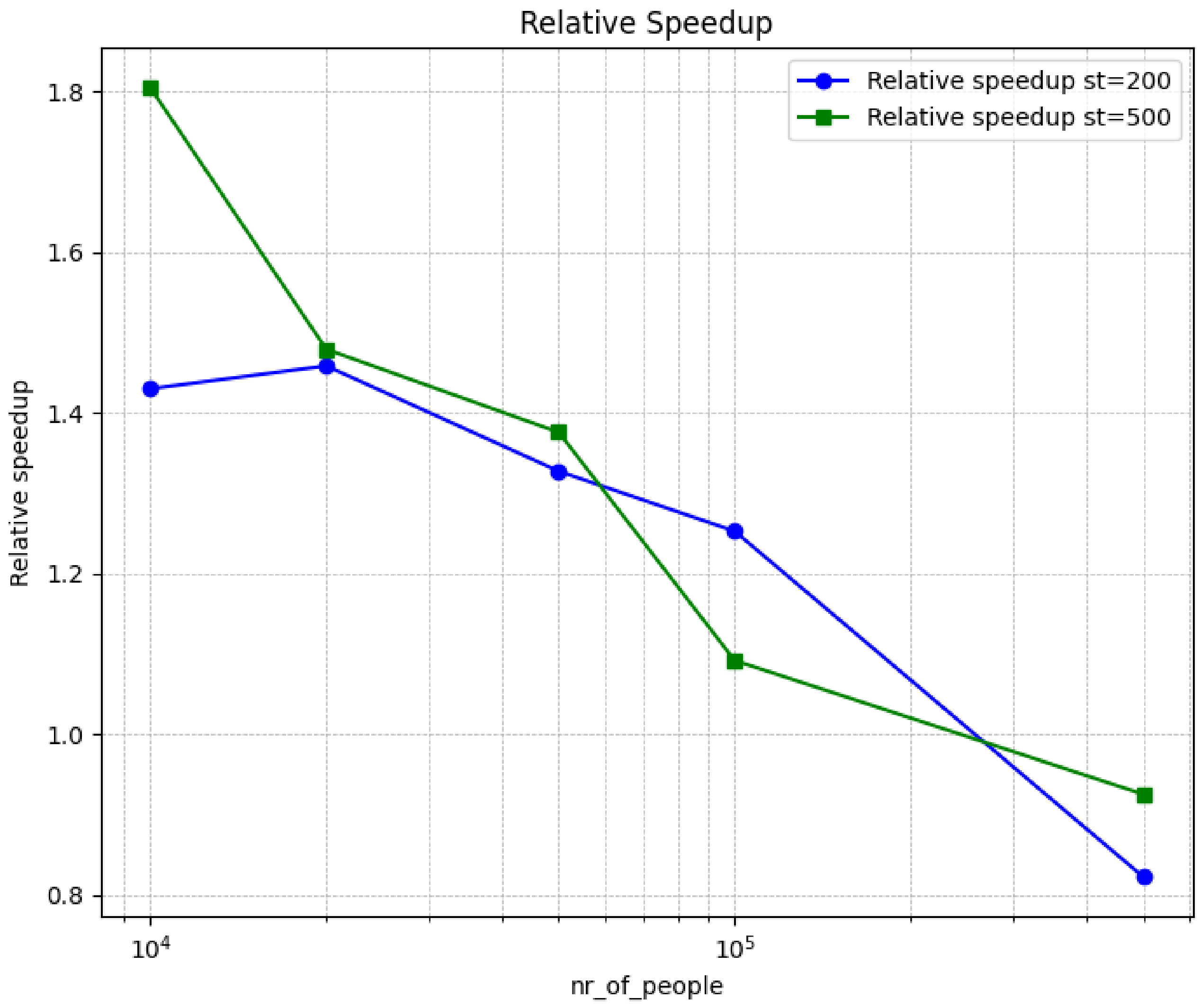
- All 3 of the implemented algorithms use the same logic to solve the problem:
  - store the input information of the people in a custom array;
  - create an integer matrix that represents our movement grid;
  - at the beginning of each iteration we parse the array and set the points on the grid where we have located at least one infected person;
  - we then parse the array a second time to determine which of the people got infected during this iteration (by checking the infection array) and in the same loop we also update the state of each person, preparing them for the next iteration.
- Details about the functions:
  - they need the flag SAVE_RESULTS defined in order to store the final results (for testing performance reasons);
  - they return the runtime of the functional part of the algorithm;
  - they provide the additional debug mode in which the state of the people is printed after each iteration (must define DEBUG at compilation);
  - we assume that both the number of people and the dimensions of the grid are divisible by the number of threads for simplicity.
- The parallel versions divide work on the people array amongst threads via:
  - omp parallel for: the ideal scheduling determined through testing is dynamic policy with chunk size 15 ;
  - omp parallel with manual data partitioning: the array is divided by the number of threads and each thread receives the entire block that it needs to operate on at start.

Project structure

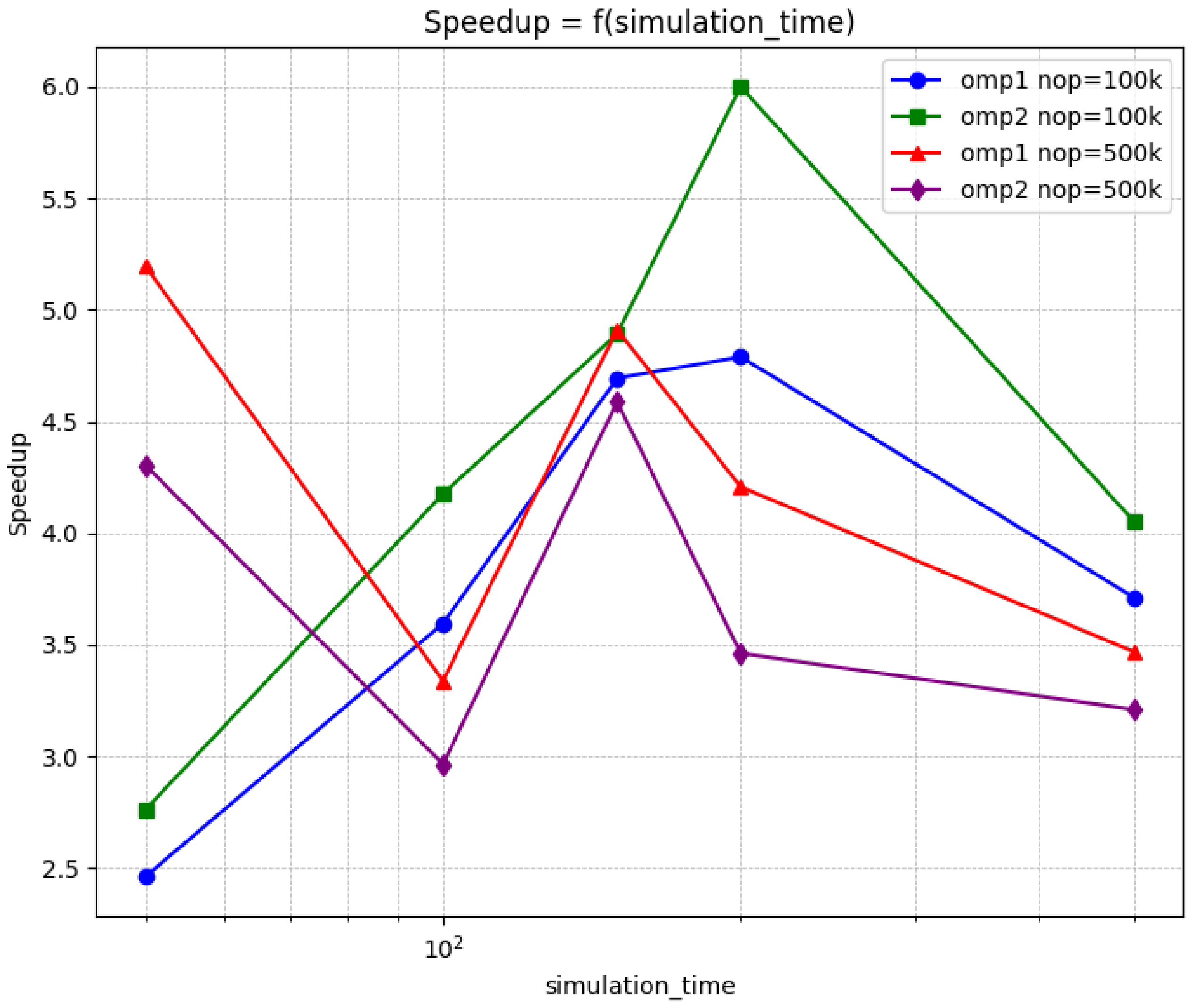
- The functions and data structures used in the epidemics simulation are defined inside of a custom C library "epidemics_utils.h"
- The simulation program is found in the "epidemics_simulation.c" file and it allows you to simulate one instance of the epidemics scenarios:
    - it takes the command line arguments <total_simulation_time> <input_file> <number_of_threads>;
    - has the SAVE_RESULTS flag defined;
    - prints the time values for the given scenario at the standard output.
- The benchmark program allows us to simulate all of the 75 given scenarios at once:
    - total_simulation_time in {50, 100, 150, 200, 500}
    - input_file in "epidemics< X >K.txt" for X in {10, 20, 50, 100, 500}
    - number_of_threads in {2, 4, 8}
    - it only stores the values of the speedups (each speedup gets its own file)
    - does not define the SAVE_RESULTS (must define at compilation but should be avoided as it introduces a lot of useless file writes) and should NOT be used in debug mode!!
- Compilation: gcc –Wall –o exec <main C source> epidemics_utils.c -fopenmp [-DDEBUG]

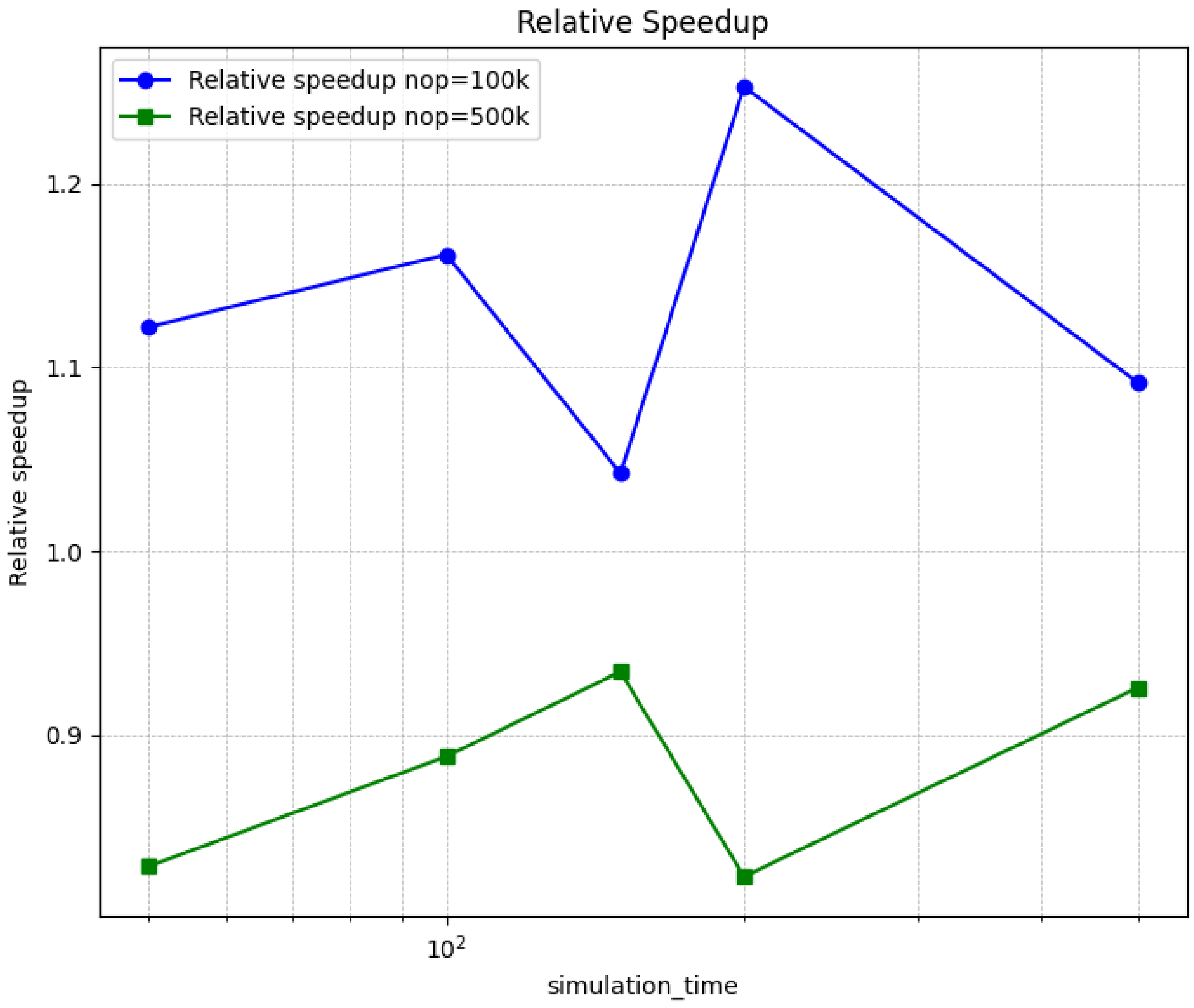
Test results1



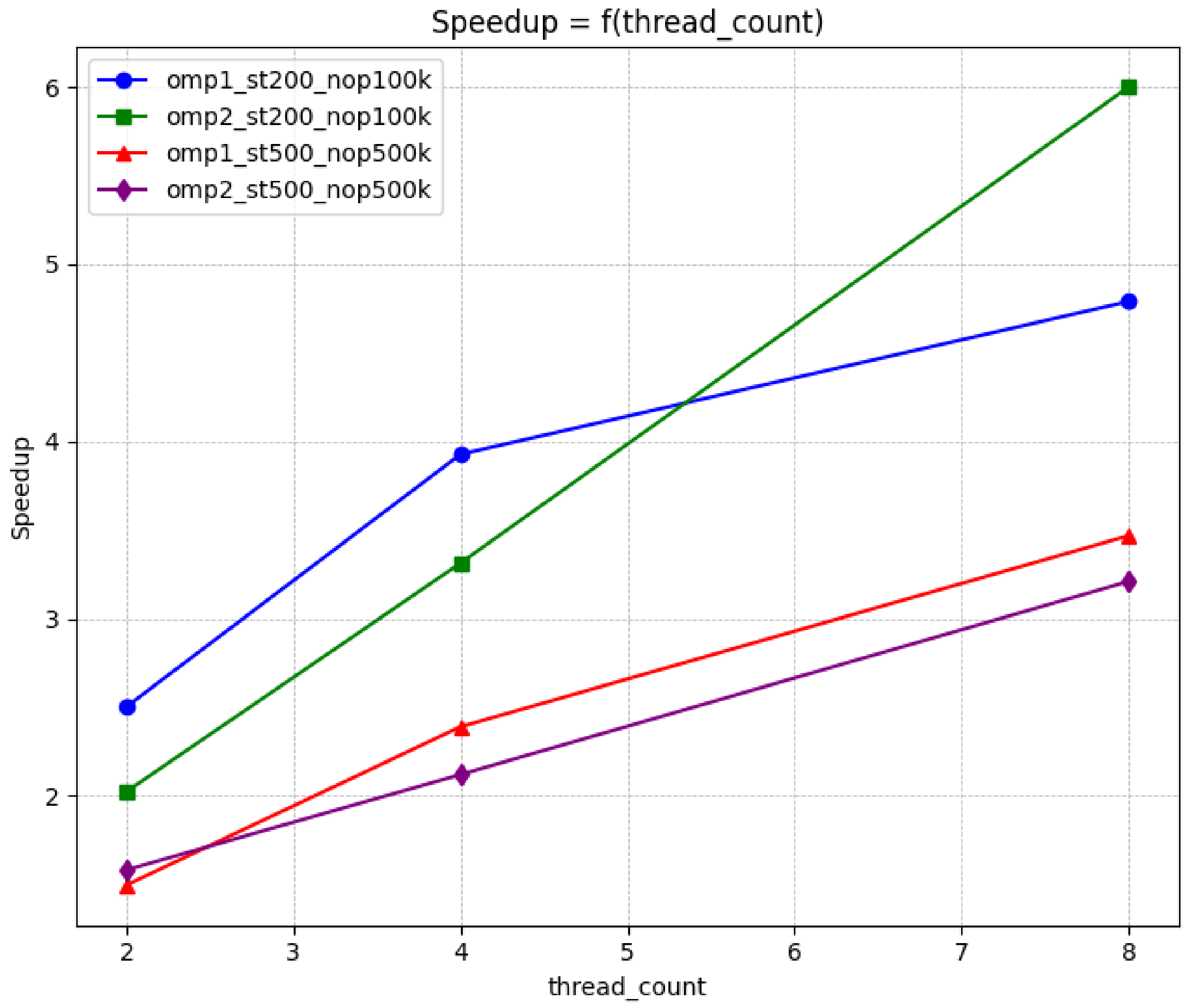


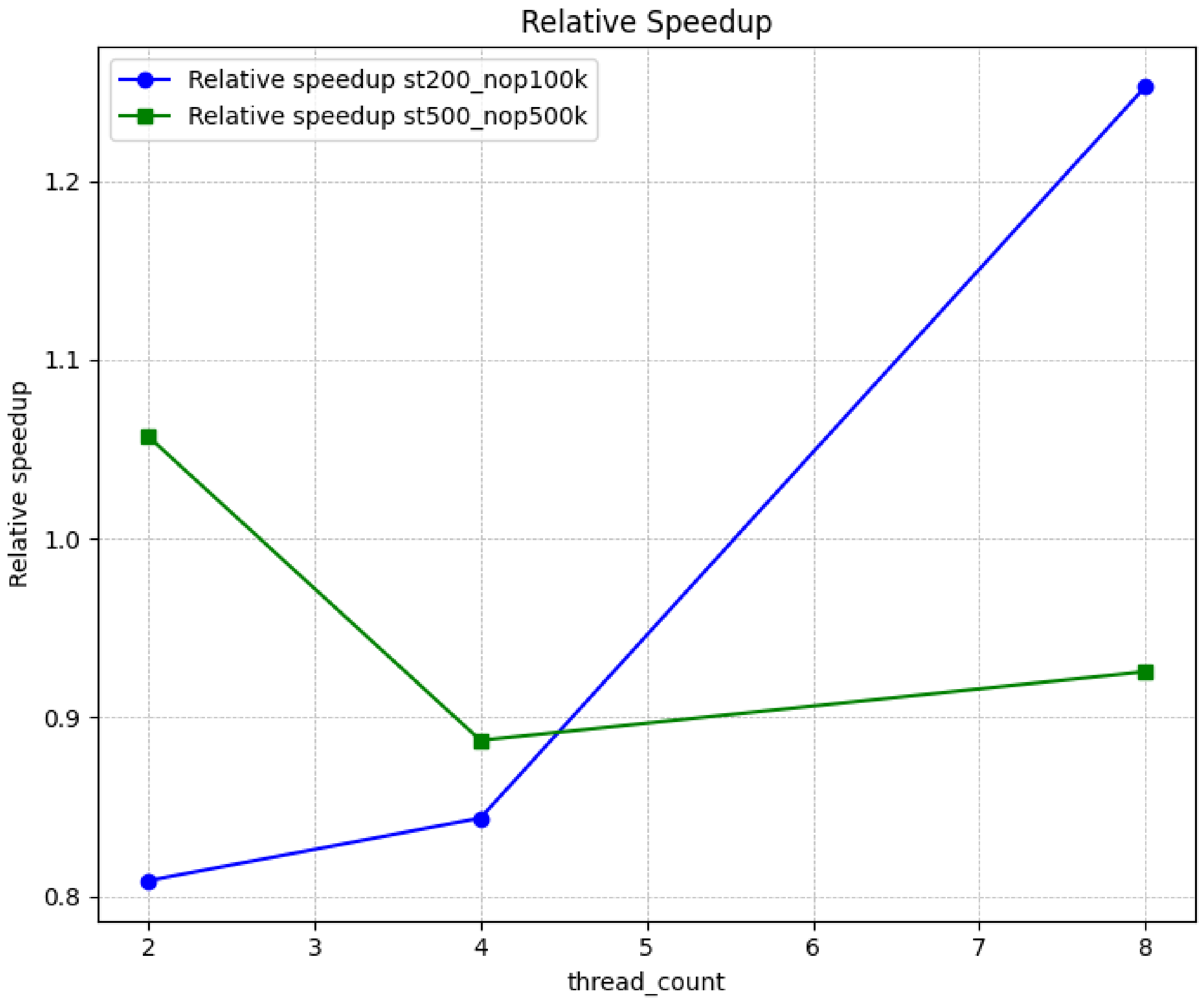
Test results2





Test results3





Conclusions1

- Neither of the parallel version is better than the other on a general level, each one of them having its scenarios where it performs better:
    - o The version with manual partitioning seems to outclass the parallel for version for a lower number of people, but its performance drops sharply at around 250K where the other version becomes better;
    - o The total simulation time does not appear to have an effect on relative performance between the two versions as the relative speedup remains rather constant;
    - o The version with manual partitioning makes better use of the threads: speedups grows linearly with the number of threads while the parallel for version seems to have a logarithmic growth;
- Performance:
    - both versions seem to peak at a population of 50K-100k and a simulation time of 150-200 iterations;
    - Manual partitioning provides a greater speedup (4.5-6) for a population under 250k but it has a quadratic decay for values greater than that (dropping to around 3.5 for my test data and likely lower for a larger population);
    - Parallel for version on the other hand provides a more stable speedup (3.5-5) which is not as affected by the increase in population size.

Conclusions2 + Limitations

- As a final conclusion the manual partitioning version is way better for a population size <250K and it makes better use of the threads, however its drop in performance for a higher population size is much greater, making the parallel for version better for a population size >250K.
- Limitations:
  - All of the speedups were only computed once as it is a lengthy process so the real graphs might look somewhat different compared to the ones displayed (improvement: run the benchmark multiple times and take the average values);
  - Only a select subset of the results were analyzed as I did not want to overload the presentation with graphs and it also requires a lot more effort to draw clear conclusions when analyzing a large amount of various data (improvement: a deeper analysis that would give us a more detailed view of how these programs perform);
  - A relatively sparce number of testing scenarios required, that leave a number of gaps in the graphic representations of the performances over which their behavior is unknown and left for assumption (improvement: increase the number of scenarios over which we test the programs with a focus on increasing the granularity of our intervals, test on a more advanced computer with a higher number of logical processors that allows for a higher number of threads).

That's it, bye :0