HOCH SCHULE OFFEN BURG

Instruction sheet for lab class 7

Exercise 1: Gauss-Seidel algorithm

Consider the 2D Gauss-Seidel algorithm as presented in the lecture. Parallelize the code with OpenMP using wavefront parallelization. Measure your speedup (compared to the original(!) serial code, not compared to the parallel code running with 1 thread).

Again, you may use the serial code to verify that your program works correctly

Which speedup do you achieve with 12 threads on the cluster?

Upload your code to moodle.

Exercise 2: Clustering with DBSCAN

We want to parallelize the popular DBSCAN algorithm for data clustering.

Please read the (very good) Wikipedia entry at https://de.wikipedia.org/wiki/DBSCAN and understand the algorithm and how it identifies clusters. You will need a thorough grasp of the algorithm in order to parallelize it.

Documentation of the existing project
In moodle, you can find a simple, serial implementation of the algorithm, closely following the pseudocode at wikipedia. Download all files.

Since this project is a little larger than the ones done so far, there are a couple of changes:

- As usual in C++ projects, interface and implementation are separated into header and source files.
- There is a makefile to facilitate working with multiple source files. If you add source files, add them at the top of the makefile so they will get compiled and linked.
- There are several make targets defined (“make <target>” will execute the commands)
    - “release” and “debug”: compile the main executable in release (fast with -O3) or debug (-g -O0) mode. Reads in data from file “data” and stores the clustered data in “clustered”.
    - “benchmark”: compile a benchmark program. Reads in data from “data” and clusters it, but does not store results.
- There are two python scripts to generate some sample data (create_data.py) and to visualize the result if the sample data is 2D (plot.py).

Task

Your task is to parallelize the algorithm with OpenMP. The parallelization is not straightforward due to data dependencies inherent to the algorithm. Think about a way, how you might achieve parallelization. Implement the fastest algorithm possible!

Verify that your parallel implementation gives the exact same results as the serial version! Use different sets of data for this, where e.g. the data is more close or further distributed. Your algorithm should work independent of the dimension of the data, e.g. with 2D, 3D or higher dimensions.

Upload your solution to moodle as a zip file. We will run a benchmark on a test data set (in more than 2D!) and compare which group has the fastest solution!
To participate, make sure that your solution can be compiled with "make benchmark" and the benchmark for 12 threads can be executed with "./dbscan_bench"!

Make a plot of runtime, speedup and efficiency as a function of the number of threads. And be honest – take the fastest serial implementation as a reference for the speedup! If you optimize something in the parallel algorithm, apply the same optimization to the serial reference.