High performance computing

Project report

Thomas Visentin (869438)

Nicolas Pietro Martignon (870034)

The problem faced

The problem considered is the application of a hierarchical clustering algorithm, to a dataset of our choice, the application of this algorithm is certainly to be considered a CPU bound task, and therefore possibly parallelizable in some of its parts in order to reduce the execution times on large datasets.

The chosen dataset

The chosen datasets, being composed of 2 features, are easily represented graphically in a Cartesian plane, therefore also by visual inspection the correlation between their representation and the dendrogram is obvious, these datasets represent data in the most disparate fields (from Gaussian distributions to coordinate on a geographical map).

The datasets on which our attention fell are in particular those reachable through this link: http://cs.joensuu.fi/sipu/datasets/

Verification of the correctness of the proposed algorithm

For doing the verification of the correctness we use the dataset “dataset2.csv” contains in the folder, which is a smaller dataset compare to the other one. Fundamental characteristic of this dataset is that every point is distant with a different value to the other points, in other words we cannot have two couple of two point that have the same distance.

The verification of the correctness of the algorithm was obtained by comparing the result obtained using the Python SciPy library (whose documentation can be found at the link https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html#scipy.cluster.hierarchy.linkage) with the result returned by our C ++ implementation of the SLINK algorithm.

The comparison takes place in the following way: starting from the distance matrix (which is obtained from the dataset) the aforementioned Python library provides the so-called "linkage-matrix" from which it is possible to easily obtain the corresponding dendrogram.

The SLINK algorithm starting from the dataset gives outputs 2 vectors (pi and lambda), these vectors can be "converted" into a "linkage-matrix" in a very simple and fast way O(n), so as to be perfectly comparable with the one that was returned from the Python library.

Obviously both matrixes obtained being the same can be graphically represented with the same dendrogram.

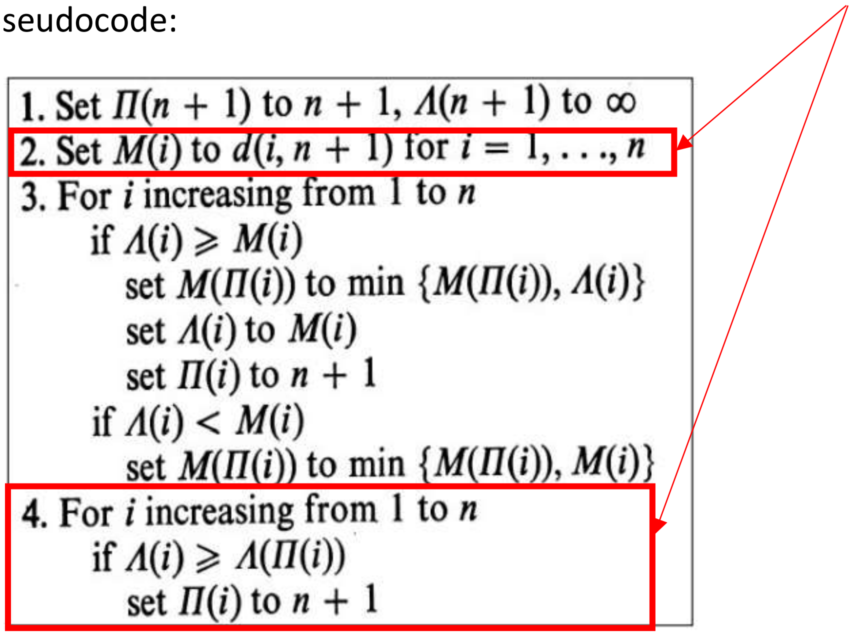
Sequential version

The sequential version is our C ++ implementation of the SLINK algorithm proposed by Sibson in 1973, this algorithm turns out to have the optimal complexity for the problem addressed in particular the time complexity is $O(N^2)$ and the space complexity O(3N), versus $O(N^3)$ and $O(N^2)$ respectively of the naive version.

In addition to the algorithm itself, other piece of code have been added, for example for reading the file and converting the result returned by SLINK (i.e. λ and π vectors) into a "linkage-matrix". This additional step of converting from SLINK to “linkage matrix” is necessary in order to compare the result obtained by Slink with the default implementation provided by the Python library as well as to graphically represent the output obtained.

Parallel version

As for the parallel version, we proceeded to divide the workload among several threads through the OpenMP library, for the parts highlighted in the following pseudocode:



```
1. Set $\Pi(n+1)$ to $n+1$, $\Lambda(n+1)$ to $\infty$
2. Set $M(i)$ to $d(i, n+1)$ for $i = 1, \ldots, n$
3. For $i$ increasing from 1 to $n$
      if $\Lambda(i) \geqslant M(i)$
         set $M(\Pi(i))$ to min $\{M(\Pi(i)), \Lambda(i)\}$
         set $\Lambda(i)$ to $M(i)$
         set $\Pi(i)$ to $n+1$
      if $\Lambda(i) < M(i)$
         set $M(\Pi(i))$ to min $\{M(\Pi(i)), M(i)\}$
4. For $i$ increasing from 1 to $n$
      if $\Lambda(i) \geqslant \Lambda(\Pi(i))$
         set $\Pi(i)$ to $n+1$
```

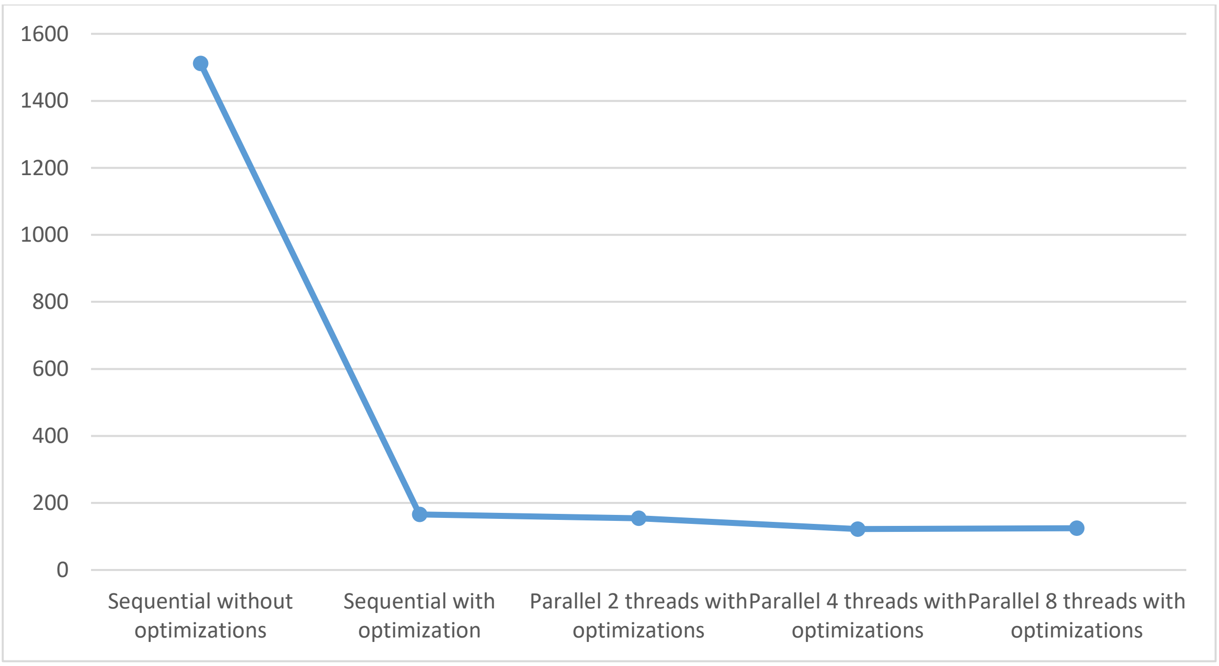
The first block of parallelized code calculates the distance from one point to other points belonging to a subset of the entire points. This subset grows until we reach the last iteration of the outer for loop, so the time elapsed for doing step 2 grow in the same way as the cardinality of the subset.

The second block of parallelized code is a "rearrangement" of the vector “PI” for the current iteration of outer for loop. Like step 2 also in this block the time needed for doing this for loop grow because every time we iterate over an always bigger subset.

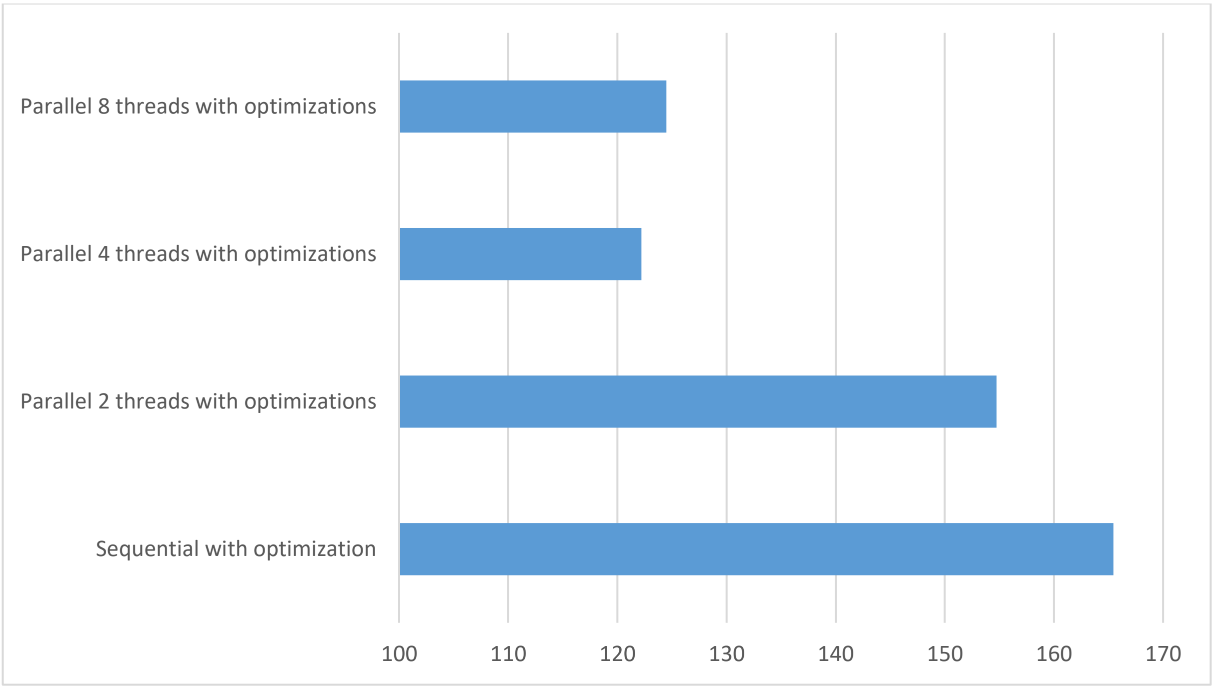
So, in these 2 steps (step 2 and step 4) there is no dependency between the variables that can prevent an effective parallelization, meanwhile in steps 3 we cannot do parallelization because for the correct execution of the algorithm we have to maintain the natural order of the for loop.

Execution times and final considerations

The first graph also visually highlights the great improvement of the times that there is simply by using the right optimization directives in the program compilation phase. The optimizations is made by compiling with “ –O3 –march=native ” options, in which compiler will vectorize the code to exploit SIMD instruction.



This second graph instead shows only a comparison between the 4 optimized versions (one sequential and three parallel), effectively providing a zoom of the previous graph.



*the graph starts from the value of 100 seconds

Speed-up: S = TS / TP   S = 166/122 = 1,36