# PARALLEL PROGRAMMING PROJECT

Graph analysis: Shortest path

## Relators:

Carlo Bobba
Eleonora Aiello

## Abstract:

The aim of this report is to show a possible implementation in the C language of two algorithms that compute the shortest path in a graph, and a possible parallelization of them using the Open MP model.

## The algorithms:

Firstly we choose the algorithms to use for computing the shortest path: we decided to use the Dijkstra's algorithm and the Bellman-Ford algorithm.
Dijkstra because is the most famous and the simplest algorithm to resolve shortest path problem, Bellman-Ford because is capable of handling graphs in which some of the edge weights are negative number, case that Dijkstra doesn't handle.

**Dijkstra** [1]**:**
The algorithm is composed of two parts
1) Initialization:
    a) Creating an array that keeps track of the vertices that have already been minimized in the distance from the source vertex and initialize its values to FALSE because at the beginning all the vertices had to be processed.
    b) Creating an array that contains the distances between the source vertex and all the others, where all the values are initialized to INFINITE except the source vertex that obviously has zero distance from itself.

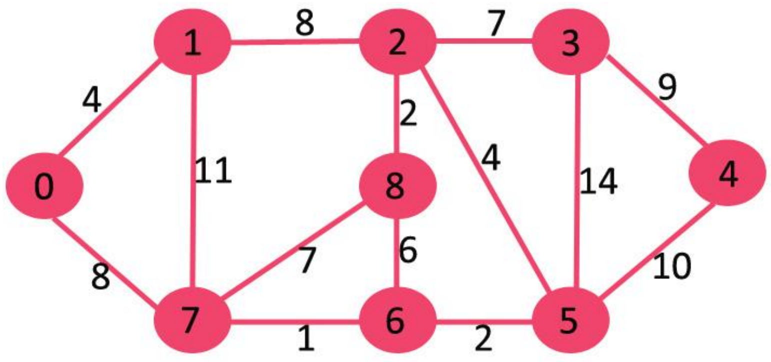
    Pseudo-code:
```
for (i from 0 to verticesNumber) {
     distances[i] = INFINITE;
     shortestPathFinalized[i] = FALSE;
}
distances[sourceVertex] = 0;
```

2) While the shortestPathFinalized have at least one vertex that isn't processed (all its values must be TRUE), the algorithm does three things:
    a. Picking the vertex with minimum distance value (which value in the shortestPathFinalized isn't yet TRUE)
    b. Setting this vertex as visited and minimized (its value in shortestPathFinalized become TRUE)
    c. Updating the distance values of the vertices adjacent to the picked vertex

   Pseudo-code:

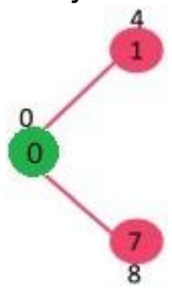
```
for (i from 0 to verticesNumber) {
    getMinDistance();
    shortestPathFinalized[minVertex] = TRUE;
    updateDistances(minVertex);
}
```

An example to clarify the behavior of the algorithm with an undirected graph is the following: having the graph below composed of nine vertices where the source vertex is 0, the minimized vertices are green and the not minimized are red.



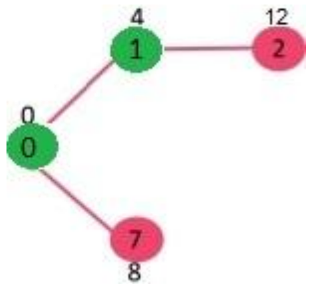
Distances = {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF stay for INFINITE
Spf = {F, F, F, F, F, F, F, F, F} where Spf stay for shortestPathFinalized, F for FALSE, T for TRUE

At the first step, I pick the vertex with the minimum distance value that is 0 and I update the distance of it to 4 and 8 of the adjacent vertices 1 and 7.



Distances = {0, 4, INF, INF, INF, INF, INF, 8, INF}
Spf = {T, F, F, F, F, F, F, F, F}

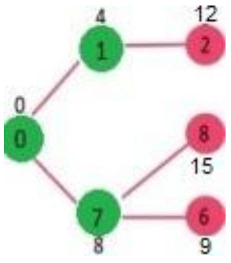
Then I pick the vertex with the minimum distance value that hasn't already been visited (its value in the shortestPathFinalized isn't TRUE), which in this case is vertex 1, I set it as visited ad update distances.



Distances = {0, 4, 12, INF, INF, INF, INF, 8, INF}

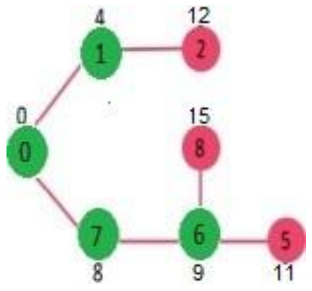
Spf = {T, T, F, F, F, F, F, F, F}

And so on, until all the vertices are visited.



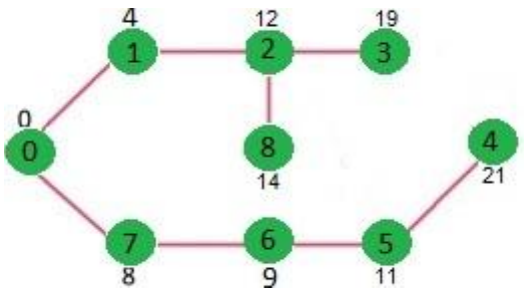
Distances = {0, 4, 12, INF, INF, INF, 9, 8, 15}

Spf = {T, T, F, F, F, F, F, T, F}



Distances = {0, 4, 12, INF, INF, 11, 9, 8, 15}

Spf = {T, T, F, F, F, F, T, T, F}



Distances = {0, 4, 12, 19, 21, 11, 9, 8, 14}

Spf = {T, T, T, T, T, T, T, T, T}

**Bellman-Ford [2]:**

The algorithm is composed of three parts:

1) Initialization:
   Creating an array that contains the distances between the source vertex and all the others, where all the values are initialized to INFINITE except the source vertex that obviously has zero distance from itself.
   Pseudo-code:

```
for (i from 0 to verticesNumber) {
     distances[i] = INFINITE;
}
distances[sourceVertices] = 0;
```

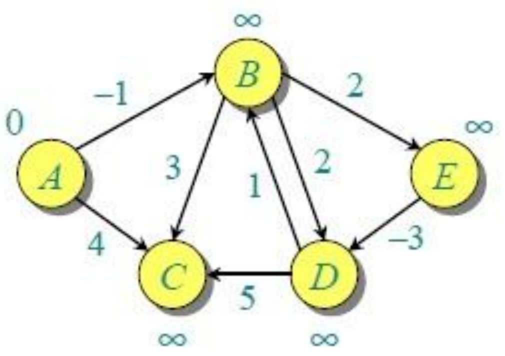
2) Relax edges repeatedly:
   An approximation of the correct distance is gradually replaced by more accurate values until the optimum solution is reached.
   Pseudo-code:

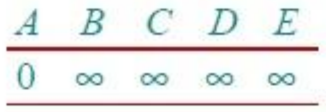
```
for (i from 0 to verticesNumber-1) {
     for each edge (u, v) with weight w{
          if (distances[u] + w < distances[v] {
               distances[v] = distances[u] + w;
          }
     }
}
```

3) Check for negative cycle presence:
   Checking the presence of negative cycles that are cycles in which the sum of the edges is a negative value.
   In that case, there is not a valid shortest path because any path become cheaper and cheaper by walking around the cycle more times.
   The algorithm **only checks** the presence of cycles and warns the user of them, it **does not manage** them.
   Pseudo-code:

```
for each edge (u, v) with weight w{
     if (distances[u] + w < distances[v] {
          printf("The graph contains negative cycles");
     }
}
```

An example to clarify the behavior of the algorithm with a directed graph is the following: having the graph below composed of five vertices where the source vertex is A, the number of relaxation to be done on all the vertices are 4.

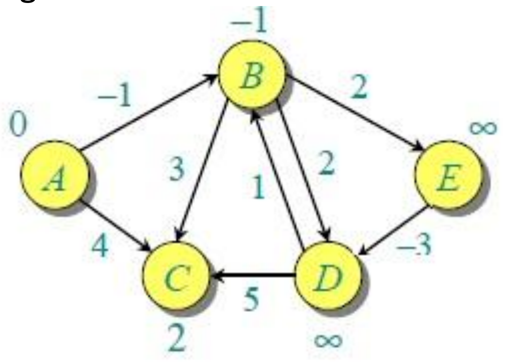


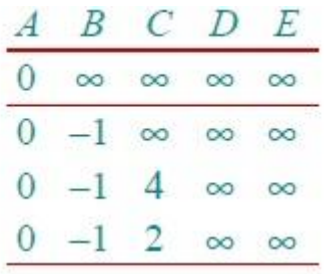


| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |

Distances = {0, INF, INF, INF, INF}

The first iteration (the first relaxation of vertices) guarantees that all vertices that are distant 1 edge from the source are minimized:

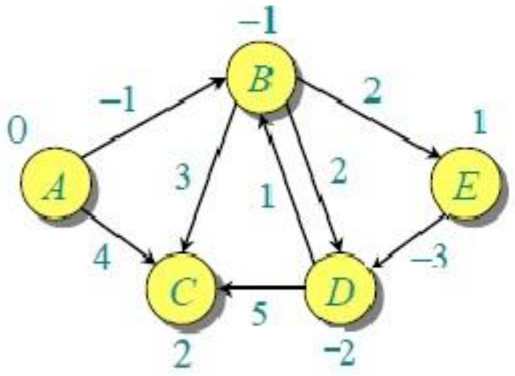


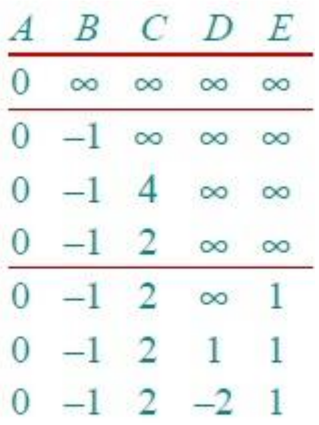


| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | −1 | ∞ | ∞ | ∞ |
| 0 | −1 | 4 | ∞ | ∞ |
| 0 | −1 | 2 | ∞ | ∞ |

Distances = {0, -1, 2, INF, INF}

The second iteration guarantees that all vertices that are distant 2 edges from the source are minimized:





| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | −1 | ∞ | ∞ | ∞ |
| 0 | −1 | 4 | ∞ | ∞ |
| 0 | −1 | 2 | ∞ | ∞ |
| 0 | −1 | 2 | ∞ | 1 |
| 0 | −1 | 2 | 1 | 1 |
| 0 | −1 | 2 | −2 | 1 |

Distances = {0, -1, 2, -2, 1}

Then the algorithm processes the edges 2 more times, also if in this case the distances computed in the second iteration are already the shortest path, so the other 2 iterations does not update the distances.

## Algorithm implementations and assumptions:

Firstly we have coded a random graph generator in order to have graphs big enough to process them with reasonable times.

Secondly, we make a function to check the correctness of the results of the parallelized algorithms. This function compares, value by value, the distances obtained by the parallel algorithm to the distances obtained by the serial one, the correctness of which was manually checked for small graph.

Furthermore, we have done some assumptions:

1) We have represented the graphs through the so-called “Adjacency matrix” instead of “Adjacency list” because it is simpler to fill it in a random way.
2) The number of edges in the graphs is defined by the “density” of the graph and more precisely, the edges are given by this formula:
$$Edges = density * (vertices^2 - vertices)$$
In our measures we decide to use a density of 75%.
3) Although the Bellman-Ford algorithm supports negative edges, we decided to avoid this case.
In fact the generation of negative edges with the random graph generator implies with high probability the presence of negative cycles, which produces non-consistent results.

The code of the two algorithms has been taken from [1] for Dijkstra and from [3] Bellman-Ford, and it is been refactored to be more efficient and readable (see the attached code).

Then, about the test designing, we have decided to examine five case of study for each algorithm, where they differ for the dimension of the graph:

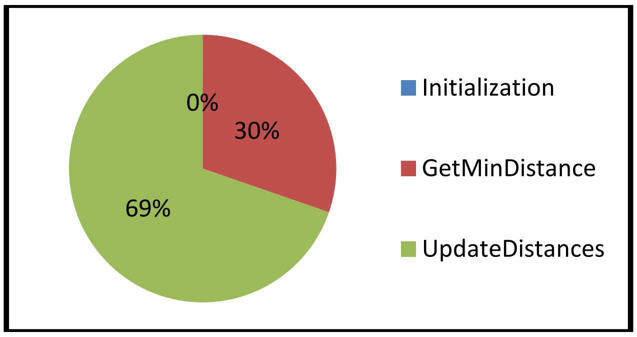
a) For Dijkstra graph dimensions are 2048x2048, 4096x4096, 8192x8192, 16384x16384 and 22000x22000.
b) For Bellman-Ford, graph dimensions are 256x256, 512x512, 1024x1024, 2048x2048, and 4096x4096.

The difference on the graph dimension is caused by the different time complexity of the two algorithms.

For each case of study, we have done five measures to care of the random nature of the generated graph structure and after that, we have taken the mean value of these measures.

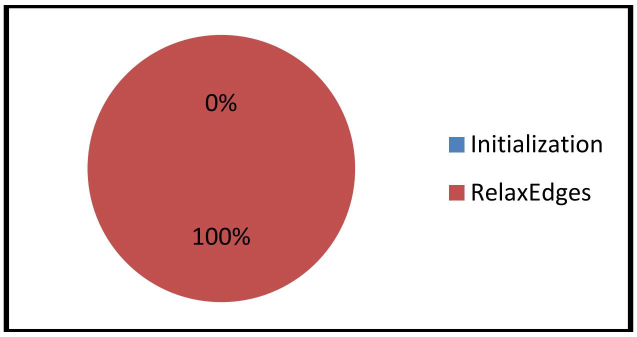
## Study of the available parallelism:

### Dijkstra:



1) Initialization:
   The time requested is almost zero but, even if the parallelization of this part is possible, it does not improve so much the needed time.
2) Algorithm execution:
   a. The time request for getting the minimum distance is about the 30% of the total time of the execution.
      This part can be parallelized (see “DijkstraParallelV2.c”) but it requires more time than the serial execution cause of synchronization, so the parallelization is useless.
   b. The time request to set the vertex with minimum distance as visited is about zero, because it is just an assignment instruction to be executed.
   c. The update distances section uses about 70% of the total time of the execution and it can be parallelized with reasonable profit.

### Bellman-Ford:



1) Initialization:
   Same consideration done before in Dijkstra’s initialization.
2) Edges relaxation:
   This part requires about the overall time of execution and it consists of three nested loops that can and must be parallelized for their temporal complexity.

## Analysis of the theoretical maximum speed-up:

To compute the maximum speed-up obtainable by the two algorithms we have had to analyze the assembly code generated by the compiler: for a better readability, we concatenated it with the correspondent C code using the command present in “AssemblyAndCMaker.bat” that must be launched in the “Release” folder of the project.

Then we have counted the number of lines that are done in the serial algorithms, the number that are done in serial and in parallel in the parallel version of the algorithms and, at the end, we have applied the Amdahl Law written in this way:

$$TheoreticalSpeedUp = \frac{SL\ in\ serial}{SL\ in\ parallel + \frac{PL\ in\ parallel}{p}}$$

Where SL stands for serial lines and PL for parallel lines and p is the number of processors seen by the operating system.

The results are viewable in the sheet “Code” and “Statistics” of “Relation data.xlsx”.

## Parallel implementation details:

The implementation of the parallelism can be seen in the attached code in “DijkstraParallelV1.c” and “BellmanFordParallel.c”.

We have focused on the use of the statement “#pragma omp for” because both the algorithms are based on for cycles.

In Dijkstra we used the statement with the default parameter (schedule = static) because the chunks have approximately the same size at every loop, instead, in Bellman-Ford we used the parameter schedule = dynamic because there is the possibility that some threads run faster than others.

## Performance analysis and speed-up:

The CPU that we have the opportunity to use to analyze the performances are all made by Intel but with different architectures and different costs:

1) Intel Core i7 2630QM: notebook CPU of Q1 2011, 2Ghz up to 2.9Ghz in TurboBoost, 4 cores with HyperThreading, 8GB of RAM DDR3-1333, with a TDP of 45W
2) Intel Core i5 430M: notebook CPU of Q1 2010, 2.26Ghz up to 2.53Ghz in TurboBoost, 2 cores with HyperThreading, 8GB of RAM DDR3-1066, with a TDP of 35W
3) Intel Atom Z3740: tablet CPU of Q3 2013, 1.33Ghz up to 1.86Ghz in TurboBoost, 4 cores, 2GB of RAM LPDDR3-1066, with a TDP of 2W
4) Intel Core i5 3470: desktop CPU of Q2 2012, 3.2Ghz up to 3.6Ghz in TurboBoost, 4 cores, 8GB of RAM DDR3-1600, with a TDP of 77W

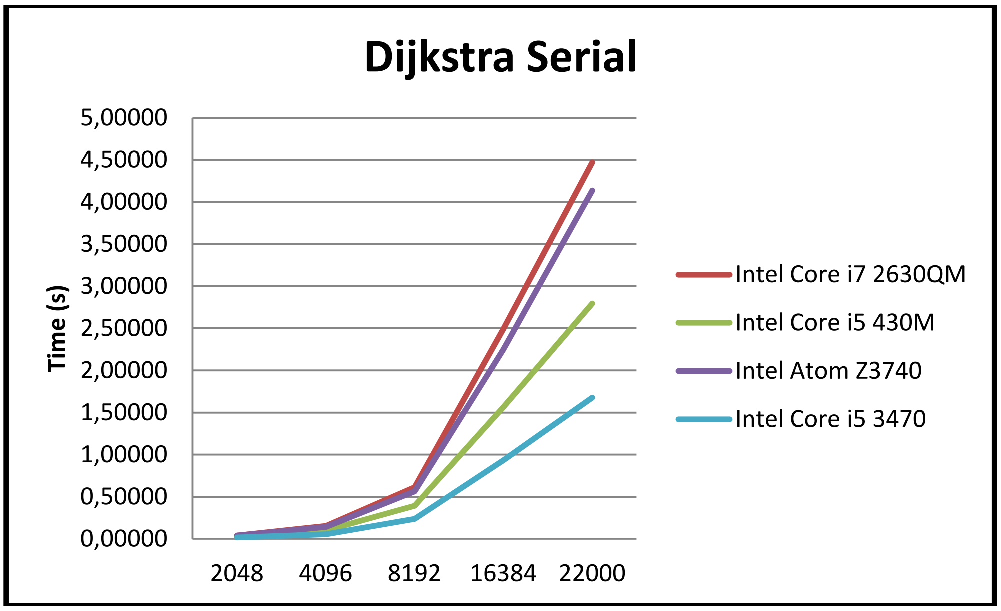
To make a performance analysis at first we have measured the time that the algorithms require for their execution on the different-sized graphs.
Then we compute the obtained speed-up with the formula:

$$ObtainedSpeedUp = \frac{Time\ for\ serial\ execution}{Time\ for\ parallel\ execution}$$

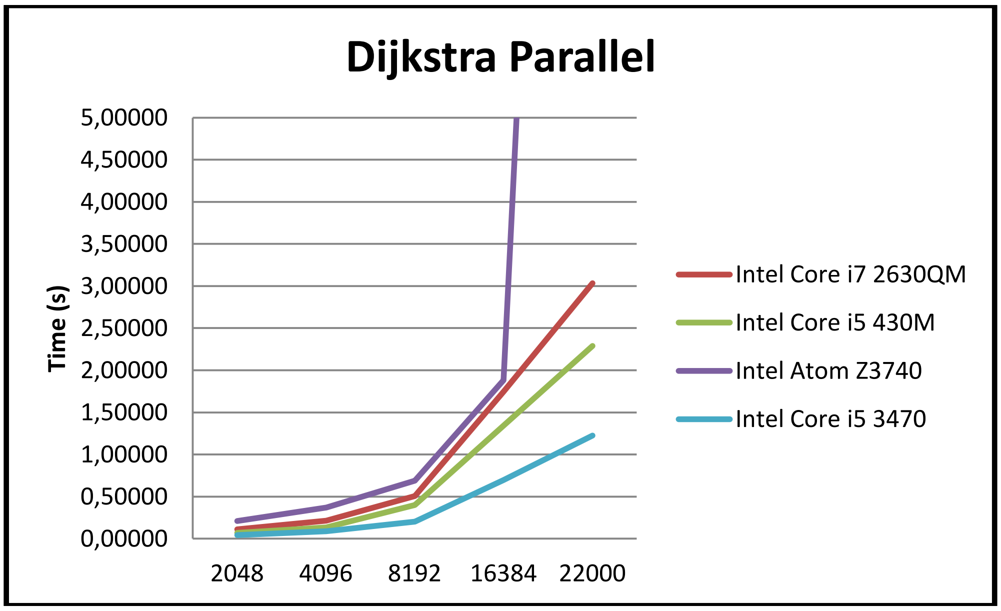
Finally, we compute the “efficiency” to make an absolute comparison between the CPUs to see how they are closer to the theoretical speed-up. The efficiency is computed in this way:

$$Efficiency = \frac{ObtainedSpeedUp}{TheoreticalSpeedUp}$$

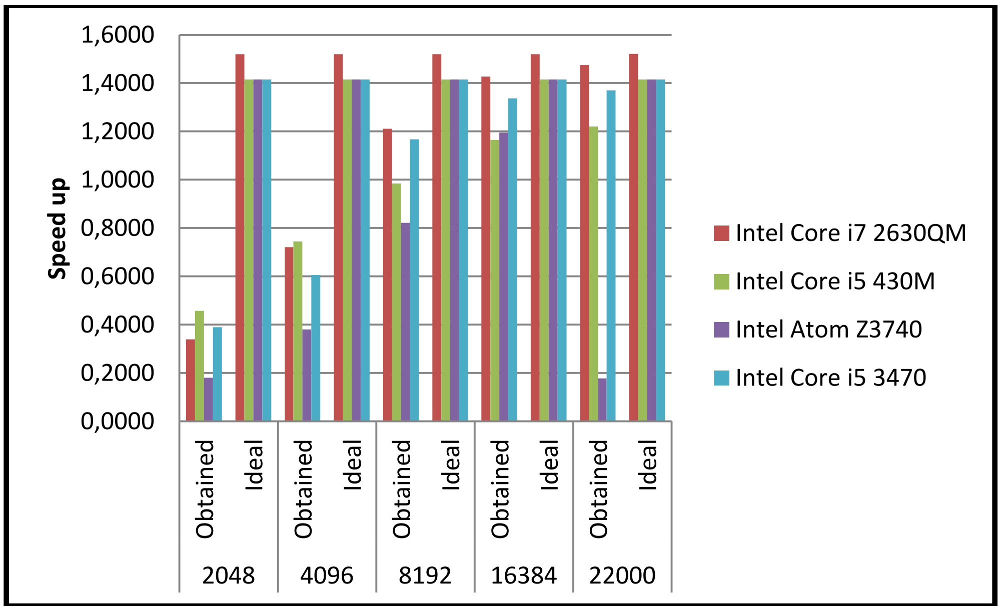
**Dijkstra execution:**
It is possible to note the fact that the time for the execution of the algorithm increase with the increasing of the dimension of the graph as we may expect.



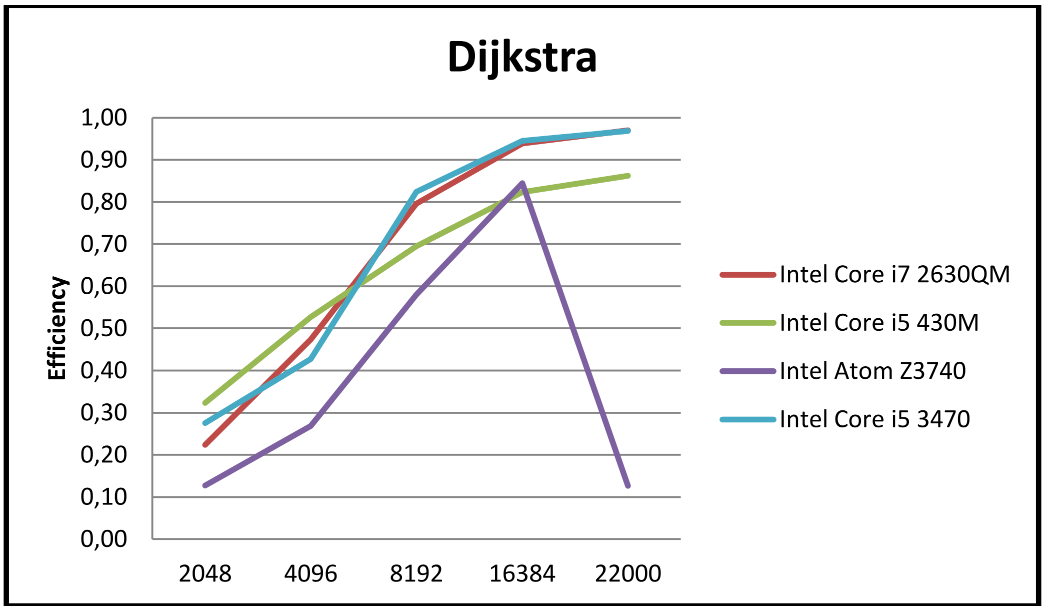
The only interesting thing happens with the Intel Atom Z3740 in the parallel version due to the tablet has only 2GB of RAM. In fact, the graph, with 22000 vertices, requires at least 1.8GB for the execution, 400-500MB are required for the operating system, so the RAM is not sufficient for the whole execution.
Consequently, there is a lot of swapping on the SSD and so the time measured is non-consistent.



At the end is possible to compute the obtained speed-up and make a comparison with the ideal computed thanks the Amdahl Law.
Naturally, it is convenient to use the parallel version of the algorithm only when the obtained speed-up is greater than 1.

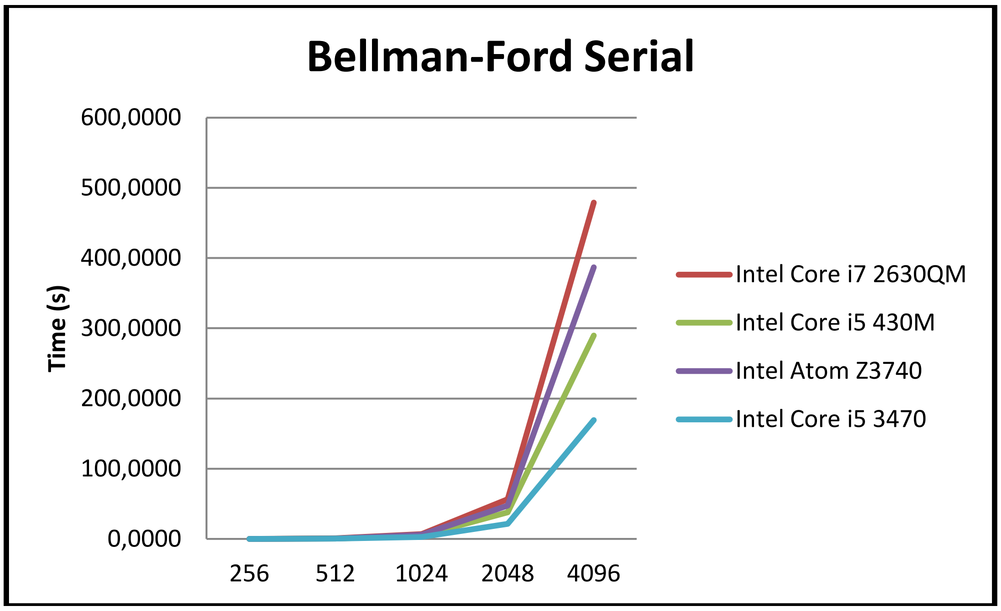


The efficiency allows seeing in a better way how the CPUs arrived closer to the ideal speed-up:

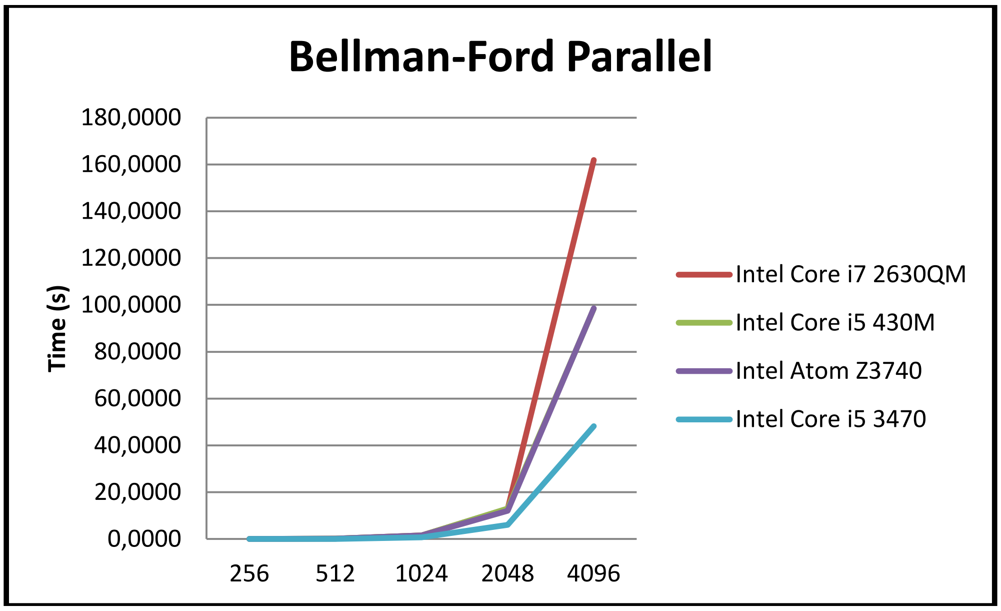


When the speed-up become consistent (generally when the graph has more than 8192 vertices), as it was presumable, the desktop CPU Intel Core i5 3470 is the more efficient, the Intel Core i7 2630QM is in the second position and the third position is of the Intel Core i5 430M. Although excluding the 22000 vertices case where the Intel Atom Z3740 is ruined by lack of memory, it is interesting to see that this is more efficient than the Intel Core i5 430M at 16384 vertices also if it has less power!

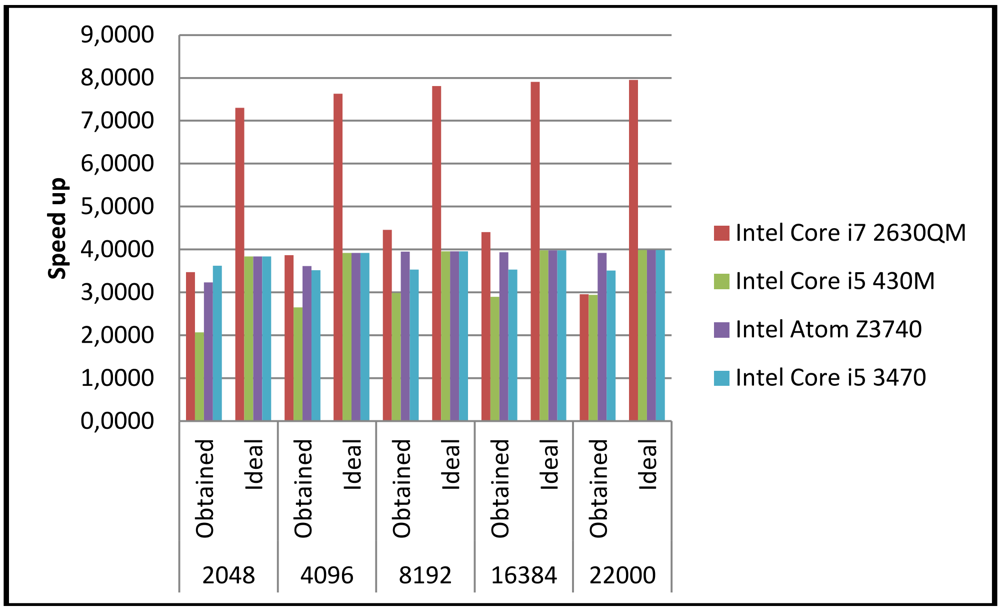
**Bellman-Ford execution:**
As seen for Dijkstra, the time for the execution of the algorithm grow up with the increasing of the graph:



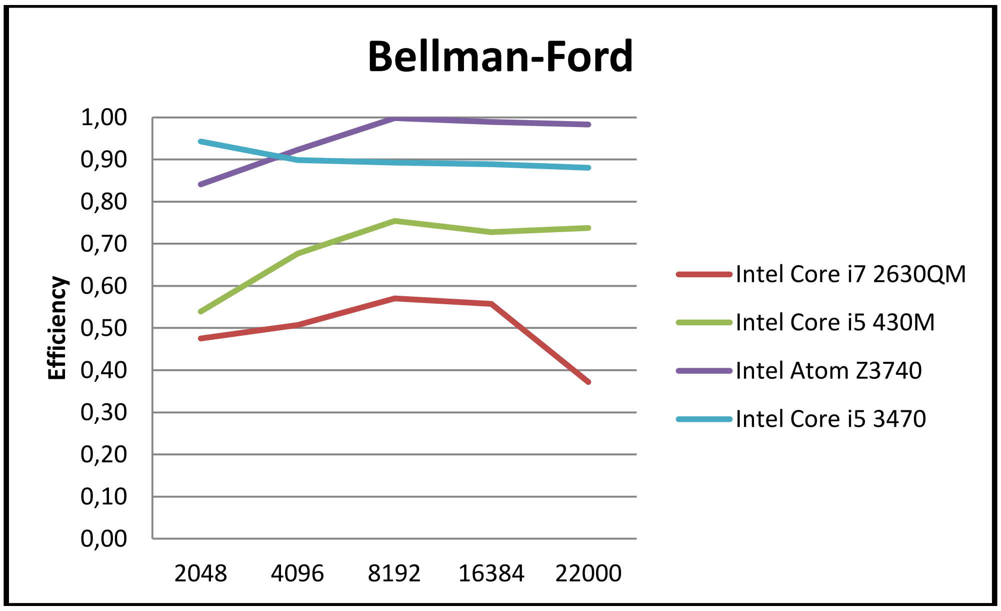
The algorithm is highly parallelizable so the time for the parallel version is smaller than the serial one:



Computing the speed-up, the fact that the algorithm is highly parallelizable is possible to see in a better way:



Consequently, analyzing the efficiency is possible to compare every CPU with the others through the efficiency:



With Bellman-Ford there are two unexpected results: the more interesting thing is that firstly the most efficient CPU is the cheapest and less powered one (Intel Atom Z3740), secondly, there is the desktop CPU (Intel Core i5 3470) that is stable at 90%. Then there is the Intel Core i5 430M that becomes stable with big graphs at 75%, and at the end, unexpected, the Intel Core i7 2630QM that have a very high loss of efficiency with the biggest graph (22000 vertices).

## Analysis conclusion:

The Intel Atom Z3740 obtains an unexpected result, because it seems to be the most optimized in the execution of parallel program, even if it is the cheapest one with the lowest energy consumption.
HyperThreading technology does not increase too much the performances, even if it depends on the CPU used. For example the Intel Core i7 2630QM has a speed up in Bellman-Ford that is about 4.5, so it looks like having one-half more physical core. While the Intel Core i5 430M has a value of speed-up of about 3, so it looks like having 1 more physical core.
In conclusion, having a glance at the CPUs features, not always what it seems to be the most powerful, it is indeed. For instance, the Intel Core i7 2630QM obtains a good speed-up in Dijkstra's, but it requires more time than the others in the execution of the code. In Bellman-Ford, it behaves as already written but also the speed-up degrades.

## Credits:

[1]
Code, images and algorithm description:
http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/
Algorithm description:
http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[2]
Images and algorithm description:
http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/
Algorithm description:
http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

[3]
Code:
http://www.sanfoundry.com/java-program-implement-bellmanford-algorithm/