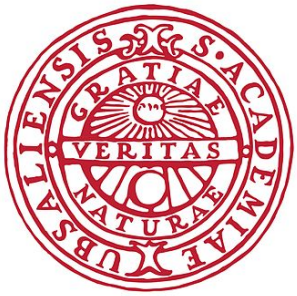

UPPSALA UNIVERSITET

Optimization and parallelization of Dijkstra's algorithm
Project in High Performance Programming

Daniel Hjelm 980406-2454
Daniel.Hjelm.0958@student.uu.se

March 1, 2022

1 Introduction

In 1956 the Dutch software engineer and computer scientest Edsger W. Dijkstra were out shopping with his fiancee and took a coffee break at a café terrace in the centre of Amsterdam. Dijkstra had for a while wondered about how you should travel between the city of Rotterdam to Groningen if you want to minimize the traveled distance. As he is an engineer, he wondered if there would be some way to design an algorithm to answer the question. While his fiancee enjoyed her cup of coffee, Dijkstra designed the algorithm without paper and pen in what he claimed to be only 20 minutes. The same algorithm was three years later published and is now one of the most famous search algorithms in the computer science field.

Dijkstra's algorithm is designed to find the shortest path between two position, which may for example represent the shortest road between two places or shortest path between two nodes in a connected graph. There are several existing variants of the algorithm which all are based on finding the shortest path. As mentioned in the previous paragraph, the algorithm was originally designed for finding the shortest road between two cities, i.e shortest route between two given position. However, the most frequently used one, oftenly referred to as single-source, uses one selected position usually referred to as source or start and finds the shortest path to all other positions in the network/graph. This version comes with regular use in applications such as digital mapping, network routing and flight scheduling.

In the context of this project, the single-source version of the algorithm will be implemented to create the so called Dijkstra's all-pairs shortest paths algorithm. This version of the algorithm finds the shortest route from every single position to all other positions and is an extension of the single-source version. The all-pairs algorithm is obtained by calling the single-source algorithm on every single position in the network and is by its nature a suitable algorithm to parallelize.

2 Solution

2.1 Adjacency matrix

In order to implement the algorithm, one must be able to represent the input graph in a suitable manner. There are several ways to do this, including for example adjacency lists, but maybe the most interpretable representation is an adjacency matrix. An adjacency adjacency matrix A is a matrix where a non-zero entry at position A_{ij} represents that position i and j are connected (adjacent). The value of the non-zero entry is the called the weight which represents the distance between the positions. By the nature of the representation, all diagonal elements will be zero since a positions distance to itself is zero. Non-connected positions is usually represented by a zero as well but to avoid confusion they are represented by a very large number in the matrix and a X when the matrix is displayed. An example of an adjacency matrix is displayed in Figure 1 below:

```
0    X    7    8    6
X    0    X    6    7
3    X    0    X   10
4    2    X    0    X
2    3    6    X    0
```

Figure 1: An example of a 5 x 5 adjacency matrix

To explain the nature of the adjacency matrix, we can take a look at the first row which represents all the connection for the first position. The first element of the row is 0 since the distance of a position to itself is zero and X in the second column means there is no connection between position 1 and 2. However, the next three elements implies that position 1 is connected with position 3, 4 and 5. The value 7 in the third column entail the distance between position 1 and 3 is 7.

2.2 Dijkstra's algorithm

The single-source Dijkstra's algorithm takes the starting node and adjacency matrix as input. In this implementation, the size of the adjacency matrix and an empty array allocated to represent the distances between the nodes in the graph is given to the function as well but it is not a requirement when implementing this type of algorithm. The function then performs the algorithm in the following manner:

1. Initialize the distance array with maximum possible number (or just a large number) except for the starting position which is set to zero. Create an array representing whether a position is visited or not and initialize it with zeros representing that no positions is visited yet.

2. Loop through every position and perform the following in each loop:

   (a) Find the position with the shortest distance and update it as visited.

   (b) For the found minimum position, loop through all other positions. For each position, check if all of the following is true:

      i. The position is not already visited

      ii. The position and the minimum position is adjacent.

      iii. The pathway from the start through the minimum position to this position is smaller than the saved distance in the distance array.

   (c) If they are all true, set the distance in the distance array to be the distance from the start position through the minimum position to the given position.

As previously stated, Dijkstra's all pairs is then achieved by calling the single-source for every single node in the graph. The efficiency of Dijkstra single-source is $\mathcal{O}(n^2)$ which implies a $\mathcal{O}(n^3)$ behaviour for all-pairs. An example of the result of the Dijkstra's all pairs is displayed in Figure 2 where adjacency matrix displayed in Figure 1 was used.

```
0    9    7    8    6
9    0   13    6    7
3   12    0   11    9
4    2   11    0    9
2    3    6    9    0
```

Figure 2: An example of the result from Dijkstra all-pairs.

In this example Dijkstra finds a path, and indeed the shortest path, for every source position in the graph. As an example, we can see that Dijkstra finds a path from position 1 to 2 with a distance of 9 (entry at first row and second column). The algorithm finds this distance by first going to last position with distance 6 (entry at first row and second column in Figure 1) and then to position 1 from the last position (entry at last row and first column in Figure 1).

2.3 Optimization

To optimize the performance of the algorithm, compiler flags is used. The -O2 flag turns on every optimization technique which does not bring about a larger executable. This proved to be a better choice than the maximum compiler optimisation -03 flag, since the code was executed faster. In addition, the -ffast math flag is used which for instance makes the code assume that all math is finite and also breaks the strict IEEE compliance. Moreover, the -march=native was used which tells the compiler to build the code so that it is native the computers CPU. Lastly, the flag -funroll-all-loops enabled loop iterations to be done in groups, also known as loop unrolling, which improved the performance even more.

In addition to the compiler flags, comparison operations in the code can be improved. In the inner loop when the adjacent position to the minimum position is updated if a certain set of statements is fulfilled, the order of the comparisons can be optimized. By first checking if the path from the start through minimum position to the current position is shorter than the current value in the distance array, the speed is increased a lot. The same type of optimization can be done when finding the minimum position were the checking if the position is already visited should be performed first.

Another optimization technique that improved the performance was to fetch the distance of the minimum position and save it to a variable prior to last innermost loop. It results in a single look up in the distance array for every position rather than looking it up several time in the inner loop.

Although not making the final version of the optimization, there were additional optimization techniques tested during the implementation. Some examples are function inlining, using keywords such as restrict and const and trying to auto-vectorize the code which all did not make the final cut due to fact that they did not contribute to any performance improvement.

2.4 Parallelization

By the nature of Dijkstra's all pairs, it can be parallelized in an easy manner. Every available process can be assigned to execute the singe-source Dijkstra to find the shortest route from a single position to all other position. Since there is no overlapping between parallel singe-source Dijkstra calls, no communication such as variable and data sharing between processess is needed. In addition, every call of single-source Dijkstra should have approximately the same load meaning there is no need to worry for load imbalance. If there is a sufficient amount of threads available and the parallelization is successful, the run time should behave like $\mathcal{O}(n^2)$.

3 Performance

In this section the performance of the unoptimized, optimized and parallel versions are evaluated for different sizes of the adjacency matrix. The test were run on Uppsala university's Linux host called Vitsippa which has the CPU AMD Opteron (Bulldozer) 6282SE (2.6 GHz, 16-core, dual socket). The test were run with all compiler flags and the compiler version was GCC 4.4.7 20120313 (Red Hat 4.4.7-23). In Table 1 below the performance of the unoptimized and optimized version is compared for a matrix size of 1000:

Run	Unoptimized	Optimized
1	4.25	3.08
2	4.24	3.08
3	4.26	3.07
4	4.30	3.09
5	4.25	3.07
Average	4.26	3.08

Table 1: A table presenting the execution time for the unoptimized and optimized implementation of Dijkstra all-pairs for an adjacency matrix of size 1000.

As seen in the results in the table, the optimization proved to significantly decrease the execution time. A 28% decrease in time on average is the impact of the optimization, indicating that it is beneficial to optimize the code more than just adding compiler flags.

Furthermore, the performance of the parallelization should be evaluated. To evalute the parallelization, the parallel implementation was run with matrix size of size 1000 on Vitsippa. In the Figure 3 below, the speed up of the parallel all-pairs Dijkstra is plotted as a function of threads:

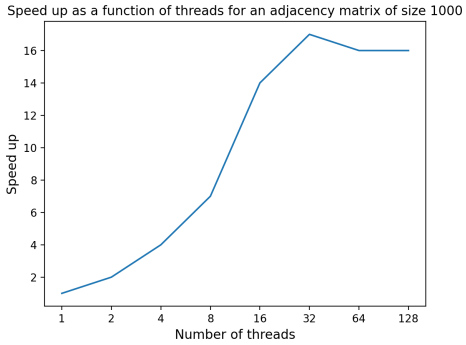

Figure 3: The speed up of the parallel implementation as a function of number threads.

From Figure 3, it is clear that the parallelization is working quite well. The speed up is approximately doubled when the number of threads are doubled up to 16 threads which is an optimal behaviour of a parallelization. However, when the number of threads increases beyond 16, the speed up begins to stop and actually decreases for 64 and 128 threads. This is the case because Vitsippa doesn't have that many threads and hence it is not beneficial to add more threads.

In addition to analyzing the execution time of Dijkstra for a matrix size of 1000, it is also interesting to analyze how the time increases when the size of the input matrix increases. To check this behaviour, the running time for both of the implementation is plotted as a function of adjacency matrix size in Figure 4 below:

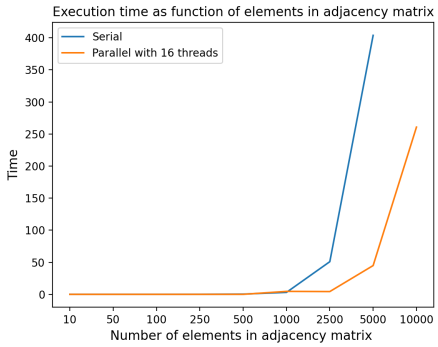

Figure 4: The execution time for the serial and parallel implementation as a function of adjacency matrix size.

From Figure 4, it is clear that the parallel version is the one to go with when the size of the input matrix increases. The serial version sky-rockets in execution time as the matrix size increases above 1000 while the parallel implementation keeps the execution time at an acceptable level. However, when the input matrix is small, say below 100, the serial version outperforms the parallel implementation marginally. This is a consequence of the time it takes to start and stop additional threads which almost takes more time than the actual calculations when the input network is small.

4 Conclusion

In this project, the Dijkstra's all-pairs shortest path algorithm have been implemented, optimized and parallelized. Optimization on top of the powerful compiler flags proved to be profitable, decreasing the running time with up to 30%. Thanks to the nature of the all-pairs Dijkstra algorithm, the parallelization is easily implemented and gives a pleasant speed up for large input matrices.

4.1 Further improvements

In the scope of this project, the parallelization of the all-pairs version was the only parallelization performed. However, there is a possibility, although is quite much more complicated, to parallelize the single-source version. This can be done using for example 1D block mapping of the adjacency matrix [1]. As a consequence, the single-source would be parallel and a parallel version of the all-pairs could be implemented on top of that, possibly improving the performance even more.

Moreover, one could express the graph using some other graph representation mehtod. One possible representation is the adjacency list where the graph is expressed as an array of linked lists [2]. The adjacency list have some advantages when storing the graph and adding new vertices. However, adjacency matrix is better for querying so it would be intresting to see if the performance would change when changing graph representation.

Lastly, Dijkstra's algorithm is not the only algorithm finding the shortest path. There are several others, including for example Bellman-Ford which can manage negative weights, A^* using a heuristic function to guide the search and another, more robust, all-pairs algorithm called Floyd-Warshall. A further improvement would be to implement the other algorithms and compare it to Dijktra's, possibly finding the best use case for each of the shortest path algorithms.

References

[1] Grama A, Gupta A, Karypis G and Kumar V. Introduction to Parallel Computing, Second Edition [Internet]. Boston: Addison-Wesley. 2003 [cited 2022-02-25]. Available from: `http://srmcse.weebly.com/uploads/8/9/0/9/8909020/introduction_to_parallel_computing_second_edition-ananth_grama..pdf`

[2] GeeksForGeeks. Comparison between Adjacency List and Adjacency Matrix representation of Graph. Last updated 2021-10-8 [cited 2022-02-25]. Available from: `https://www.geeksforgeeks.org/comparison-between-adjacency-list-and-adjacency-matrix-representation-of-graph/`

[3] Wikipedia contributors. Shortest path problem. Wikipedia, The Free Encyclopedia. Last updated 2022-01-15 [cited 2022-02-25]. Available from: `https://en.wikipedia.org/wiki/Shortest_path_problem`