Dijkstra algorithm, getting the shortest path in a graph

Luis Ricardo Montes Gomez
ID: 153788
luis.montesgz@udlap.mx

Octavio Palomeque Gasperin
ID: 151884
octavio.palomequegn@udlap.mx

March 23, 2019

1 Abstract

Dijkstra algorithm is useful when itâs necessary reach from one state to another with the lowest cost, there are many implementations of it. We had developed one that search the shortest path from a source node to any other. Then we get the times for 6,8,15,16,26,32,64,128,256 nodes and we plot them using Octave.

2 Introduction

In this document we are going to talk about Dijkstra Algorithm to try to parallelize it in order to topics seen in class. This algorithm is an implementation to find the shortest path between two nodes in a graph, this path is based in decisions taken with the information of each vertex weight that connects each node in the graph, in other words, the problem can be explain as follows:

Dijkstra algorithm is an implementation to find the shortest path between two nodes in a graph, it was proposed by Edsger W. Dijkstra in 1956. The original algorithm just find the shortest path between a node A and node B, but there are many variations of it, the most popular is one that finds all the shortest paths between a source node and any other node in the graph, producing a

shortest-path tree.

We need a directed graph $G = V, E$ with weights and V having the set of vertices. The special vertex s in V, where s is the source and let for any edge e in E, EdgeCost(e) be the length of edge e. All the weights in the graph should be non-negative.

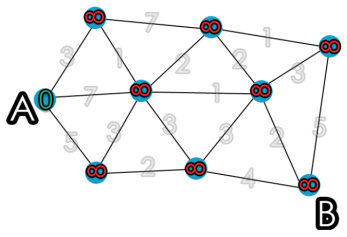

Figure 2: First Iteration.

In the Figure 3 we can see how the algorithm set the cost of travel from A to their neighbors.This is the first iteration , thatâs the reason that the intersections visited had the value of cost without any other value added.

3 Problem Description

The first step is set all intersections with âinfiniteâ just to note that this intersection has not been visited yet, then, at the first iteration, we look for the closest intersection and relabel the intersection with the weight of travel through that edge, and so on with the other intersections, then we move to the closest intersection and look for the closest intersection, if it isnât âinfiniteâ we check if the current intersection weight plus the cost of travel to that intersection is less than the current cost, if that is true we relabel the intersection with the sum and continue with the other intersections. With Figure 2 we illustrate how the algorithm works.

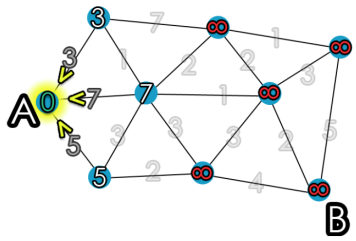

Figure 3: Second Iteration.

As can be seen, in Image 4 the algorithm move to the intersection with the lowest cost. After that itâs necessary to visit the neighbors intersections and label it with the actual intersection value plus the cost from travel to it. If we performed the previous operation we get one intersection with 4 and other with 10.

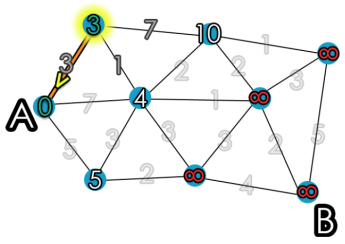

Figure 4: Fourth Iteration

The Image 5 shows the final result, after visit all the intersections and do the correct comparisons to determine if the path that we are following is really the shortest or the one with the lowest cost. Thatâs the reason because after get one result following the lowest costs, is necessary to return and verify what happen if we follow another path. Like if we go from âAâ to the intersection with a cost of five, then if we performed the sum of costs it return a greater cost from one intersection and the same value from the other one. The latest means that if we travel from that path the cost to achieve to the goal will be more expensive.

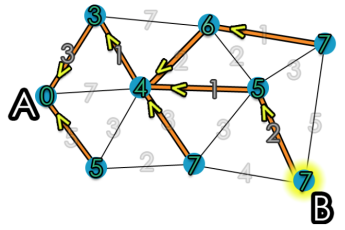

Figure 5: End of algorithm

4 Complexity of the algorithm

To calculate the Dijkstra algorithm complexity can be count based on the number of actions required to find a certain solution based in some conditions. The algorithm consist in $n-1$ iterations as maximum. In each iteration you add a vertex in to the set. In each iteration, the vertex with the lowest tag among those that are not in S_k must be identified, the number of this operations is bounded by $n-1$. Finally you have to perform a sum and a comparison to update the tag of each one of the vertex that are not in Sk_2. After each iteration are done at most $2(n-1)$ operations. The Dijkstra algorithm makes $O(n^2)$ operations to determine the longitude of the shortest path between two nodes of a simple graph. The complexity of the algorithm then could be defined by the next formula in case that we are not using a priority queue $O(|V|^2 + |A|)$

5 Procedural implementation

5.1 Algorithm Description

For the procedural implementation we used the basic algorithm for the solution. As it could be different with distributed solutions, in the procedural implementation we just have to follow one path, that is why the solution is pretty simple. As you can see in the next pseudo code we will describe step by step the way we implemented it.

The first thing that we have to do is initialize all the nodes with an "infinite" distance and initialize the starting node with a 0 and at the same time we will mark this distance as permanent. After that what we did is to calculate all the temporary distances of all neighbours nodes of the active node by summing up its distance with the weights of the edges. If this distance that we calculated is smaller as the current one, we update the distance and set the current node as antecessor. This is the main idea of Dijkstra and finally we have to set the node with the minimal temporary distance as active. Finally we have to run these steps until there aren't any nodes left with a permanent distance

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        dist[v] := infinity
        previous[v] := undefined
    dist[source] := 0
    Q := the set of all nodes in Graph
    while Q is not empty:          // main loop
        u := node in Q with smallest dist[ ]
        remove u from Q
        for each neighbor v of u:
            alt := dist[u] + dist_between(u, v)
            if alt < dist[v]       // Relax (u,v)
                dist[v] := alt
                previous[v] := u
    return previous[ ]
```

5.2 Results of the procedural implementation

As can be seen in Figure 7 the time of execution increase while the number of nodes increase, this incrementation is linear so as more nodes you add into your problem to solve more time consumption will require. In a posterior section we will comment how this differs from the shared memory implementation and the distributed memory implementation.

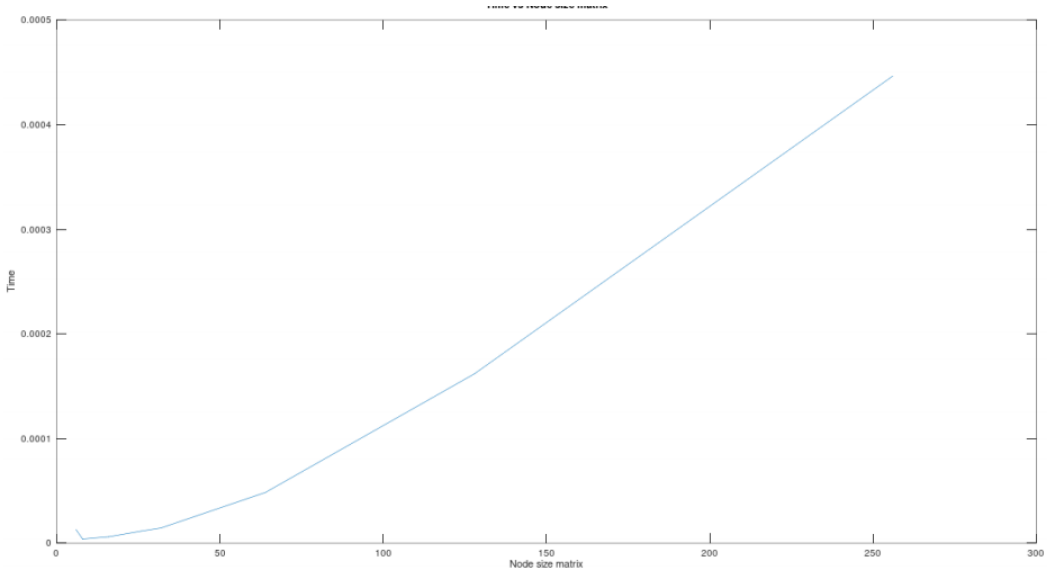

Figure 6: Results after executing the code

6 Shared Memory Implementation

6.1 Algorithm Description

As we saw in class for the parallel implementation of the Dijkstra problem we used the library called OpenMP inherent to gcc, using a technique that is included in this library called pragmas that give the instruction to the compiler to send several tasks in the implementation to separated processors in the computer giving the opportunity to reduce the time consumption in relation to how many instructions are possible to be parallelized, this is what we call scalability.

In addition to this, is important to mention that our implementation is parallelized in an automatic way, which adds the parallelization in every sector that is possible, the most common one is in the cycles of the code.

Using 2 simple instructions into the code, when the compiler compiles the code adding the instruction to use openMP it will start using several processors to do the task.

```
#include <omp.h>

#pragma omp parallel num_threads(4)
#pragma omp for
```

6.2 Results of the shared memory implementation

The parallel implementation shows that just with some simple implementation of pragmas the efficiency of the algorithm increase considerably good. The reduction is considerable from the first processors to the second one, therefore as you can see in the figure 9, even if the addition of a second processor into the analysis creates a reduction of time consumption is not the same with a third or fourth processor because there's no more threads to paralleliced. In other words, the scalability of the problem got to the top of the possibilities, in other circumstances or other kind of problems where more parallelization possibilities can be applies this could increase with a third or more processors.

7 Distributed Memory Implementation

7.1 Algorithm Description

The first thing that we have to do when we are trying to do the distributed memory implementation of our algorithm is to decompose our problem in several sub-problems. It is not something trivial, it is not clear how is constituted by independant parts the algorithm.

We start making the division between the nodes, in each part of our program where a loop for all nodes is given we will restructure our scheme to consider only a subset of the nodes.

When we are using MPI the way the development of the algorithm works is completely different and at the same time is required to use a complete new deployment of instructions based in the MPI libray.
The first thing that we have to do is define a communicator, this elements is like a box that will be the one in charge of contain and determine which process are involved in the communication in what moment and after that initialize the MPI communication between the process.

After that we have to define a number of nodes that will be taken for each process based on the rank that we will decide defined by the equation $loc_n = N/P$ being loc_n the elements taken for each processor and then we will create spaces in the disk that will take this information into consideration to define the spaces that we will create to store the matrix, the distances and the predecessors.

Later on we have to define using the properties of MPI a channal of 'communication' taking in consideration the kind of elements that we will manage defining and building a block of data with specific properties that are the ones that will be communicating.

Finally we have two more steps that we have to do, the first one is to read the matrix and assign to each process the nodes that will be doing and once each process have their nodes assigned after applying an initialization function for Dijkstra it has to start doing the algorithm that will compute all the shortest paths from the source to all vertices and then we will obtain the minimum distances obtained by each process. And the last thing that we have to do is to apply a gathering function that will take all the answers obtained and put them together in the root process to get a final result from all the individual solutions.

7.2 Results of the distributed memory implementation

The distributed memory implementation shows an improvement in the time execution, however because we have a data dependence factor in our solution doesn't give us a solution where the more processes you use a better result you will get, also we take into consideration that it would take more time execution for the communication. When the numbers of processors is even the performance grow but when we increase the number to 4 the speed decrease. In the next section you can see a better evaluation with a graph

8 Comparison between the implementations (Procedural, shared memory and distributed memory implementations)

As we can see in the next figures, the execution time of the algorithm grows as you add more nodes into the execution, starting with 50 and finishing with 500, the execution time is almost linear, therefore, once we add the pragmas into the implementation as we can see in the figure 9 the time execution is reduced making it incredible faster as it was with the procedural implementation.

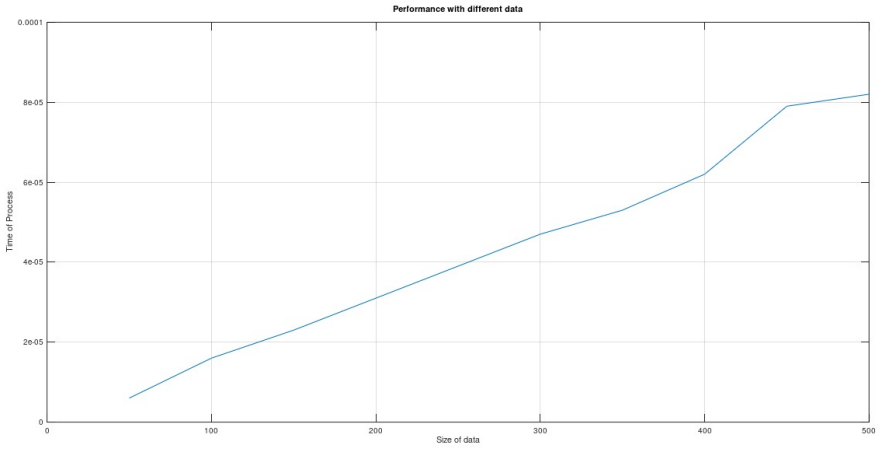

Figure 7: Chart that shows the execution time based in the amount of nodes in the graph.

The addition of the pragmas into our program shows us a big difference between the procedural and the parallel implementation, and this method used was quite simple to incorporate into our implementation. So the relation between the benefit that we got and the human-time required for the addition is incredible effective.

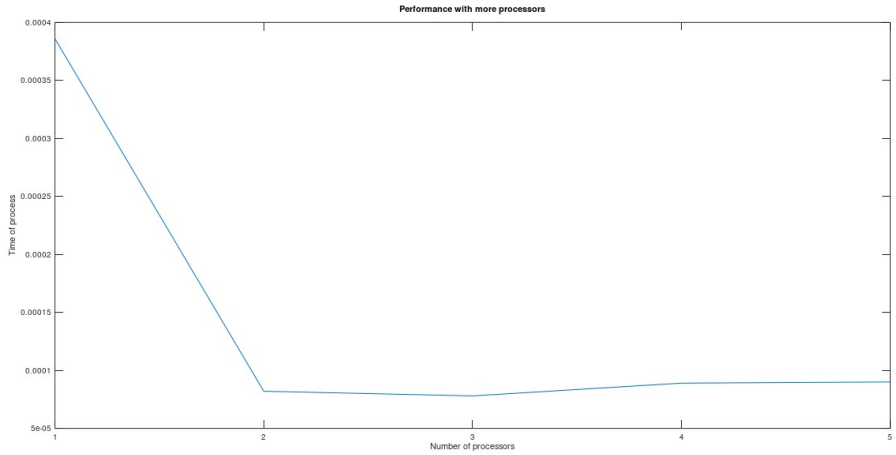

Figure 8: Chart showing the reduction in time execution after the parallelization of the algorithm.

Adding a new element in to the comparision we have the distributed memory solution, based on the results that we obtained we conclude that the most efficient one is the distributed memory solution, therefore, that doesn't mean that more processes will give us a better result because the more processes you use more time communication you will require so it is necessary to determine how many processors are the best result based on the time execution as you can see in the figure

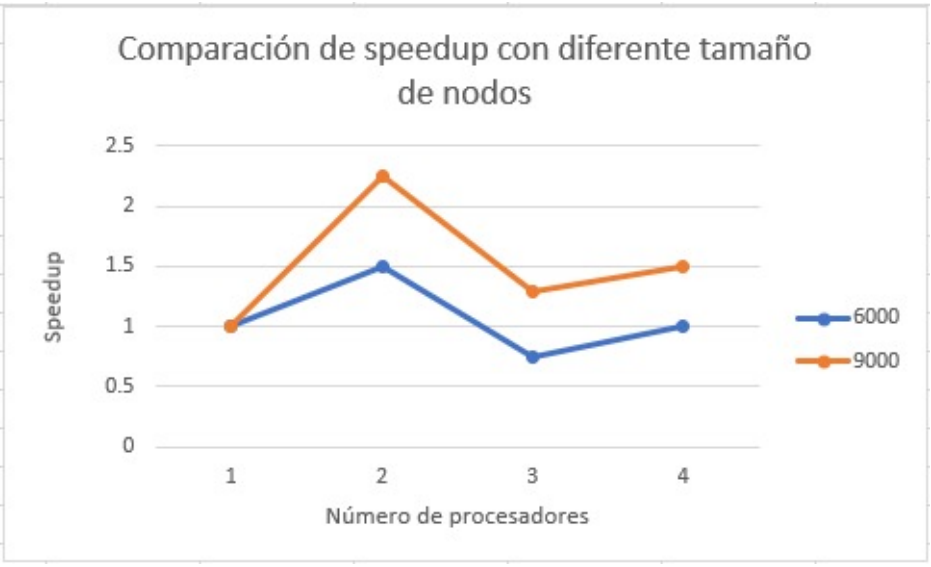

Figure 9: Chart demostrating how the speedup is executed based in the numbers of processes

9 Conclusions

After test with different data the Dijkstra Algorithm we can say that it's a good way to search the minimum path between two nodes, because the execution time it's not to high. For the main porpoise of this implementation, the algorithm is a good one to make it parallel, because it have many for loops and the search trough the paths could be faster if one processor search in different paths.

After we had the opportunity to do the implementation with parallel programming we obtained the results that we expected, after the implementation we saw an improvement in the time execution as is showed in the figure 9 that we saw previously.

Based on the results that we obtained using MPI to develop our parallel implementation we conclude that the data dependence of this algorithm could be the fact that cause the low speedup reached. As can be seen the speed performance grows when the number of processors is even. But when we increase the number of processors to 4 the speed decrease, this could be for the communication between the processors.

A positive fact is that the performance for large data using MPI increase a lot in comparison with the OMP implementation or with the sequential one.

Finally, we would like to add that as we saw in class, most of the applications don't take into consideration the use of multiple compute power from even as something as it could be a laptop, the lack of this implementation in several programs and applications create a waste of compute power making that most of the time we are just using one processors when is simple to add an automatic parallelization.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *I*ntroduction to algorithms. MIT press.

[2] Brandes, U. (2001). A faster algorithm for betweenness centrality. *J*ournal of mathematical sociology, 25(2), 163-177.

[3] Johnson, D. B. (1973). A note on Dijkstra's shortest path algorithm. *J*ournal of the ACM (JACM), 20(3), 385-388.

[4] Goodrich, M. T., Tamassia, R. (2002). *A*lgorithm design. Wiely India.

[5] GÃ©za VÃ¡rady, BogdÃ¡n ZavÃ¡lnij (2014). Introduction to MPI by examples. *h*ttps://www.tankonyvtar.hu/en/tartalom/tamop412A/2011-$0063_2 3_i ntroduction_m pi/ar01s07.html$