Flight Distance Computation: Distributed Single Source Shortest Path Implementation

Pedro Valente, Pedro Camponês
Departamento de Informática
FCT-UNL
Portugal
{pm.valente,p.campones}@campus.fct.unl.pt

Abstract—A distributed architecture possesses very good scalability properties, as its total computing power can be increased by adding independent components. The increase in computing power achieved by adding more machines to the system may be offset by the increase in communication costs incurred. As such the use of a distributed architecture for solving computationally expensive problems must consider the size and characteristics of the problem, else the performance may degrade.
In this project the use of a distributed architecture is explored in the computation of *Single Source Shortest Path* problems over a data set containing information regarding distances between airports. Two algorithms are implemented; one to be used locally and one which can be used either locally or in a distributed setting. The performances of these algorithms are compared with varying problem sizes and configurations.

1 Introduction

A distributed system is composed of several machines physically separated. Distributed computing is indefinitely scalable, being able to perform increasingly harder computations as more components are added. In a shared memory architecture increasing a machine's performance requires expensive components and is limited by the components' concurrent accesses to memory, particularly when modifying memory locations accessed by several components [5].

In a distributed architecture adding components is comparatively inexpensive, and because components are independent, if one suffers a failure the system continues to work properly [2]. Despite having the presented advantages over shared memory architectures, distributed architectures are not ideal because of the incurred communication costs between independent components. As the number of independent components increases, so does the communication cost. These costs limit the scalability of the system; this bound is dependent on the requirements of the problems being solved and how efficiently the implementation uses the properties of the distributed architecture.

As the amount of data that needs processing increases [4], it is necessary to take advantage of distributed systems' scalability and fault tolerance properties to tackle hard problems. *Apache Spark* [1] is a unified analytic engine that allows the processing of data on physically separate machines, abstracting most of the communication from the programmer. In this project, *Apache Spark* is used to compute distances between any two airports based on a data set containing information concerning a series of airplane flights.

The implementation using *Spark* is then compared with a sequential implementation on a single machine that incurs neither the benefits nor the drawbacks of a distributed architecture. With this, the performance gains with the use of *Spark* are measured as the size of the program varies.

The remaining report is structured as follows: Section 2 describes the information contained in the flights data set, how it will be processed, and the challenges it presents for a distributed implementation.

Sections 3 and 4 present the sequential and distributed implementations to solve the problem, respectively. In section 5 the relative performance of both implementations are analysed. The report is concluded in section 6.

2 Problem Description

The aim of this project is to identify the fastest route between two airports given a data set containing information about airplane flights. Each flight has, among other attributes, an airport of origin and one of destination, a departure time, and an arrival time. With this information it's possible to estimate the average time cost of travelling directly between two airports. It is possible that there is no direct flight between two airports, or the flight is very long; in such situations the path between two airports may involve hops through a series of airports. This problem can be modelled as a graph where the airports are the nodes and the edges are the connections between every two airports with a direct flight. The cost of each edge is the average flight time between the airports. With this representation it's possible to identify the distance between every two airports using a *Single Source Shortest Path* algorithm [3].

These algorithms compute the shortest distance between every node, starting from a specific origin. To do so, and independently of the algorithm used, a structure that maintains the distance of every node to the origin is necessary. Henceforth this structure will be referred as *Shortest Distance*

Using *Spark*, the graph will be partitioned such that each machine running the program will retain information concerning the adjacencies of a limited number of nodes. The computations performed by each machine will determine the shortest path to every node; however, to maintain the program's correctness, the partitions need an updated view of the previously mentioned structure.

Maintaining an updated view of the *Shortest Distance* incurs communication costs between the machines, as every change one makes must take effect on the remaining. The challenge of a distributed implementation of a *Single Source Shortest Path* (SSSP) algorithm is thus minimizing the communication costs of maintaining the structure's view across machines, doing so implies maximizing the amount of computation each partition can perform independently and concurrently before needing to get an updated view.

3 Dijkstra's Algorithm

The simplest and most widely known SSSP algorithm is Dijkstra's [3]. The algorithm iteratively traverses every node in the graph to determine its shortest distance to the origin. On every iteration the algorithm determines the final and minimal distance of a single node to the origin, and explores its frontier. The node explored is the closest to the origin that hasn't been explored; the first node explored is the origin itself, then its closest node, followed by the second closest, etc. until all nodes are explored.

By exploring nodes only once, and by fixing one's distance at every iteration, the Dijkstra's algorithm performs the lowest amount of work of every SSSP algorithm. However, because only one node is explored per iteration, the span of the algorithm is very large, as the only parallelization opportunity is in the exploration of a node's frontier. Unless the node has a very high degree, the costs of launching several threads to parallelize the frontier exploration are greater than the benefits attained.

Because Dijkstra's algorithm has minimal work costs, it has been chosen as the sequential version of the algorithm against which the distributed version is compared. The limited amount of parallelization Dijkstra supports prevents it to be used as the distributed implementation.

4 Johnson's Algorithm

The Johnson's algorithm is a modified version of the Dijkstra's that allows the exploration of several nodes per iteration. Unlike Dijkstra's algorithm, Johnson's may need to explore nodes more than once. This property of the algorithm induces it to have to perform more work than Dijkstra's; however, the span of the program is potentially much smaller than Dijkstra's, especially when nodes have an high average degree.

In Johnson's algorithm, all nodes whose distance to the origin have been updated in the previous iteration are explored in the following. In the first iteration, only the origin is explored; the following iteration explores nodes in the origin's frontier; the third explores the frontier of the nodes explored in the previous iteration, plus nodes with an edge connecting to the origin, but whose shortest path is not along that edge. Subsequent iterations follow this logic until there is no node whose distance is left to update.

In Johnson's algorithm, every node is potentially explored a large number of times, making this algorithm intractable in the worst case for large graphs. In practice however, with the data set used, most paths between airports are either direct or require a single intermediary hop. Few paths require two hops, and three intermediary hops are rare; for instance, using *SAN* as origin, *PSG* is the only airport with a path of length four.

In Dijkstra's algorithm, the number of iterations required is equal to the number of nodes in the graph. For Johnson's however, the number of iterations is equal to the length of the longest path. In the data set provided, Dijkstra requires *310* iterations while Johnson's algorithm requires only four iterations, when having *SAN* as the source.

For the distributed implementation, every machine will manage a set of nodes. Among these machines, one will be the *master*, which will disseminate updated information to the remaining machines and aggregate the combined computations of the system; this, besides doing the work the remaining machines are tasked. At the start of every iteration of Johnson's algorithm, the *master* disseminates an updated view of the *Shortest Distance* as well as the set of nodes to be explored in this iteration.

Upon receiving this information, each machine will compute the conjunction of the nodes in its partition with the set of nodes to be updated; the resulting set of nodes will be explored by that machine. For every node to be updated, its frontier is traversed and the distance to the of the outgoing extremity of the edge nodes are compared to that of the updated *Shortest Distance*. Whenever a distance shorter than that which figures *Shortest Distance* is identified, the new shortest distance is stored and the node is marked to be updated in the next iteration.

When every machine has finished exploring its nodes, this information sent to the *master*, which will then update the *Shortest Distance* structure with the new distances computed in this iteration. On the last iteration, no node will be marked to be updated on the next iteration and the algorithm ends.

5 Experimental Analysis

To test the relative performance of the implementations a data set generator has been used, allowing the configuration of the graph independently of real flight data sets. With this approach it's possible run the algorithms in graphs with varying number of nodes and with different average degrees[1]; allowing the exploration of the effects of these parameters in the algorithms performance.

The presented experiments were run in a cluster with three machines, each with *16* hardware threads.

Besides the use of Dijkstra's algorithm sequentially and Johnson's on a distributed architecture, it has also been tested the effectiveness of Johnson's algorithm in a local setting. This version is often the most effective, as the communication costs are minimized and a high degree of parallelism is still possible using the hardware threads available on the local machine.

1. The provided algorithm to generate graphs has been modified. Whilst the given algorithm generates, for each node, a maximum of *100* edges, depending on a parameter given; the new algorithm matches every two nodes in the graph and decides whether to generate an edge based on the connectivity passed as parameter.

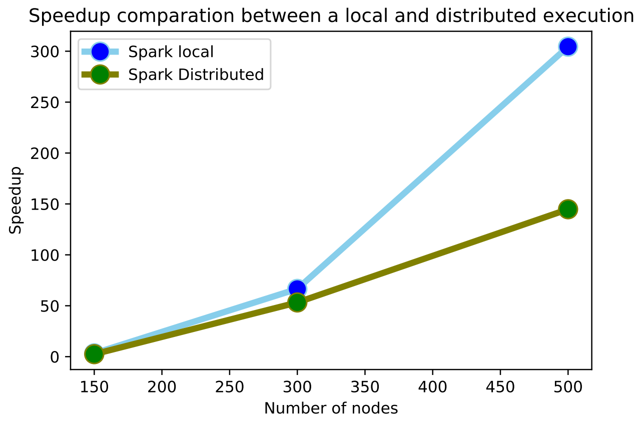

Fig. 1: Johnson's algorithm speedups with varying graph nodes

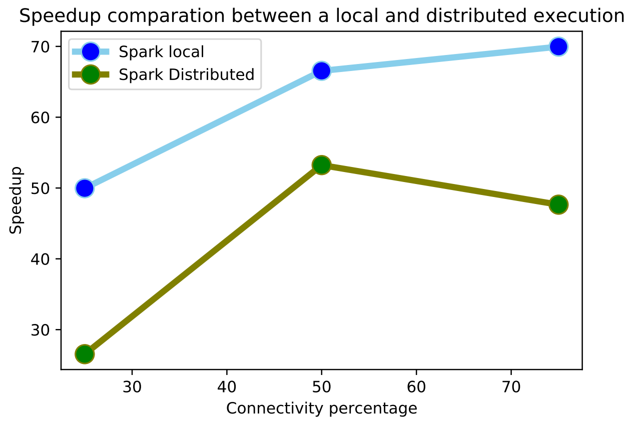

Fig. 2: Johnson's algorithm speedups as the connectivity of the graph varies

Fig. 1 presents the speedups attained using Johnson's algorithm with recourse to *Spark* in a local and distributed setting, for a graph connectivity of *50%*. Dijkstra's algorithm is heavily affected by the number of nodes in the graph, relative to Johnson's. As the number of nodes increases, so does the degree of the nodes in the graph, which in turn results in more parallel work that can be computed per iteration. At *150* nodes, Dijkstra's algorithm's performance is comparable with Johnson's, as the latter's parallel capabilities do not offset the lower amount of work Dijkstra's performs plus the inherent cost of launching threads to perform parallel computation. As the number of nodes increases, these costs are dwarfed by the parallelization capabilities of Johnson's.

Between *150* and *300* nodes, the performance of both deployments is similar, however, upon hitting the *300* nodes threshold, the local version achieves much greater performance than the distributed. As the number of nodes increases, so does the cost of communication in the distributed setting; for every iteration, the *Shortest Distances* structure must be broadcast and the updated distances must send to the *master* machine. With *500* nodes, the amount of computation required per iteration can be efficiently processed by the *16* hardware threads of a single machine; the use of *48* threads of the three machines is not efficient for a problem of this size, and so the slight performance improvements resulting from the use of more threads do not offset the communication costs, when compared with a local deployment of Johnson's.

In Johnson's algorithm every node is explored at least once, as such, assuming most nodes are explored a small number of times, the complexity of the algorithm is $O(n^2)$ with n being the number of nodes. Given Johnson's low amount of iterations, it can be considered that these have a complexity of $O(n^2)$. In practice, this has been observed to be true for the second and third iterations of Johnson's algorithm; the first iteration only explores the origin and as such has a complexity $O(n)$, similarly, iterations following the third explore a small number of nodes, such that the complexity of these iterations can be considered $O(n)$.

The communication costs incurred by each iteration grows linearly with the number of nodes in the graph[2]. For a very high number of nodes the higher computational complexity of an iteration compensates the communication costs between machines. Despite fig. 1 showing a divergence in the attained speedups between the local and distributed version, and favouring the latter; this trend is bound to be reversed with a further increase in the number of nodes.

Unfortunately, it has not been possible to compare the performance of both algorithm for very high number of nodes, as the generation of the graph proved too slow and occupied more memory than that which was supported by a single machine in the cluster. Much like the SSSP protocols being experimented, the complexity of generating a graph is $O(n^2)$ both in time as in space.

Fig. 2 presents the speedups achieved by Johnson's algorithm when compared with Dijkstra's using *300* nodes as the connectivity of the graph varies from *20%* to *80%*.

Dijkstra's algorithm's performance is not as severely affected by the general graph connectivity as Johnson's algorithm; Dijkstra's performs as many iterations as nodes there are in the graph, and while the connectivity of the graph may determine how much work is done on an iteration these are inexpensive when compared with Johnson's algorithm. The lower the connectivity of the graph, the more iterations are performed by the Johnson's algorithm, and these are very expensive in communication costs. For a connectivity of *10%*, the number of iterations performed by Johnson's algorithm is very high, which explains its its low speedups on a distributed setting. The local version of Johnson's also performs several iterations, however, it does not incur the latency costs the distributed version does.

The high speedups achieved by the local Johnson's algorithm when compared with Dijkstra's on *10%* connectivity show that even for such a low connectivity the amount of parallel work to be done each iteration is considerable.

Between *10* and *50%* connectivity, the speedups of both versions increase at a fast rate. The increase in connectivity results in a lower number of iterations to be performed;

2. The current implementation of the Johnson's algorithm does not reduce repeated modifications to the distance of a node within a partition. This increases the amount of communication incurred, an unnecessary overhead caused by an oversight.

these results are exacerbated in the distributed implementation as each of its iterations carries a considerable cost.

Between *50* and *80%* connectivity, the difference in speedups changes little, which indicates that a connectivity of *50%* is enough to ensure most nodes are connected to the origin through short paths. As the connectivity increases, the smaller is its effect on the algorithm's performance.

An interesting phenomena can be seen when considering that the local version of Johnson's algorithm's speedups changed little in the last interval, while in the distributed deployment these lowered. When the number of paths between nodes increases, so does the probability that a path previously thought to be the shortest. When this scenario occurs, an additional iteration of the algorithm may have to be undertook. So, counter intuitively, for a very high connectivity, the number of iterations of Johnson's algorithm may increase. For the local version of Johnson's a small increase in the number of iterations does not affect the speedups achieved significantly.

To conclude the observations, it can be gathered that in a graph with a high number of nodes and high connectivity, Johnson's algorithm shines when compared with Dijkstra's. The high connectivity ensures a low number of iterations to be performed by Johnson's, with each iteration having a large amount of work that can be done in parallel.

6 Conclusion

In this project, it has been presented the problem of solving the *Single Source Shortest Path* graph problem in a distributed setting, applied in practice to a data set with information on flight routes between airports spread throughout the world. The project distributed solution has been implemented with recourse to *Apache Spark*'s analytics engine, abstracting most of communication process between replicas.

It's been shown that this problem is not trivial as it depends not only on the algorithms used and the size of the problem, but also the configuration of problem's graph. Even for a large number of nodes, an unfavourable configuration may increase the communication costs of a distributed solution rendering it wasteful when compared with a simpler local solution.

Three scenarios were tested and corresponding performances compared: a sequential implementation of Dijkstra's algorithm, a local deployment of Johnson's algorithm with limited parallel potential, and a distributed deployment of Johnson's algorithm with high parallel computational capabilities but also very high communication costs.

With this project, we are able to use the *Spark* engine to tackle large problems with in a distributed architecture, understanding also whether or not such an attempt should be made by analysing the problem's properties and determining (whether through analytical or empirical observation) if it can be efficiently implemented on distributed setting.

As future work, further experiments on the effects of the graph configuration on the algorithms' performance could be explored. The graph generator used in the experimental setting attributes random weights to the edges, as such most shortest paths between two airports are direct if these are extremities of an edge. If the costs were to be squared the probability the direct path is the shortest lowers. Doing so would increase the number of nodes that are explored more than once by the Johnson's algorithm, allowing testing the algorithm in less favourable graphs.

References

[1] Apache spark webpage. Accessed 17/12/2020.
[2] Cachin, C., Guerraoui, R., and Rodrigues, L. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* 01 2011.
[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 2 ed. The MIT Press, 2001.
[4] Guo, H., Wang, L., Chen, F., and Liang, D. Scientific big data and digital earth. *Chinese Science Bulletin (Chinese Version) 59* (12 2014), 1047.
[5] McCool, M., Reinders, J., and Robison, A. *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.