Parallelization of Bellman-Ford algorithm to calculate the shortest path

Jamil Gurbanzade
University of Tartu

Abstract

This article is dedicated to describing the effects of parallelization on the Bellman-Ford algorithm that finds the shortest path from a single source vertex to all of the other vertices in a weighted digraph via “Numba” open-source JIT compiler based on Python. All in all, the consequences depicted that by running the algorithm in the parallel environment we obtain decisive efficiency in comparison with them sequentially.

Introduction

A graph is considered a pair $G=(V,E)$ where $V$ is a set of elements called vertices and $E$ is the set of vertices and called edges. In daily life the applications of graphs can be noticed everywhere. Traffic simulation, Google Map, Internet Routing planning and etc. One of the profound examples of graph applications is routing protocols that routers utilize them in data transmission. The large-scale network is typically not directly connected between one and the other, but it has gone through some other network devices. Bellman-Ford is one of the best route search algorithms that use the value of each existing link and it is used in the RIP (Routing Information Protocol) routing protocol, the hop value of each router is in its past. [1]. Most of the applications possess the negative weight of edges. Bellman-Ford algorithm is very suitable on this sort of graph that finds the shortest path even the graph has negative edges. The importance of the performance of applications comes first due to working in real-time and it is vital for them to be modified and developed. One of the efficient ways to deal with computation time is running the algorithm in parallel environments. The parallelization of the current algorithm is by adding certain constraints to the graph and extracting the hardware capabilities. This method involves the use of hardware accelerators, such as the graphics processing unit.[2]. GPU has the ability to trigger its huge parallelism in order to speed up parallel calculations of high levels. Google Colab has provided GPU solutions to run algorithms in a faster way which was used in this paper.

Related works

Davidson’s paper [3] was proposed a dynamic algorithm for SSSP (Single Source Shortest Path) problem. Due to the replicas of some vertices, there was some superfluous work and to cope with its Workfront Sweep solution was proposed with the queue data structure.

Yefim Dinitz and Roten Itzhak [4] proposed a combination of Bellman-Ford and Dijkstra algorithms that perfectly operates in the graphs where the distribution of negative edges was sparse.

In Meyer and Sanders [5] paper another efficient algorithm “Delta $\Delta$” was introduced. The algorithm possessed the tunable parameter delta. If delta equals 1 it returns the Dijkstra variant however, if the delta was taken as infinity ($\Delta=\infty$) it yields the Bellman-Ford method. Tuning the parameter between 1 and $\infty$ we get different processing time and parallelism.
The Bellman-Ford algorithm

For calculating the optimized path from the beginning vertices on a weighted graph, the Bellman-Ford algorithm is used. Unlike the famous Dijkstra algorithm, it can handle if the edges have negative weights. The pseudocode is given below for the algorithm:

![Algorithm 1](image)

Figure 1. The pseudocode for Bellman-Ford algorithm

During the relaxation process, when the edge is relaxed \((u, v)\), it checks whether it starts from the source node, and it can also update the distance if necessary.

Time complexity is evaluated in the best and worst case in this algorithm. The best-case scenario is each processor takes constant time for initializing a single node weight. Afterward, the value of the source node takes constant time. It takes \(O(n)\) time to relax for the first iteration. The total time complexity for the best case performance is \(O(V|E)\) time to relax till the final edges.

In the worst-case scenario, two first iterations take similar time to the best time, and all other iterations take as much of \(O(n)\) as possible to relax all possible edges of a vertex with \(n\) processors. Thus, the total time complexity of the best case scenario is \(O(n^2)\).[6]

The relaxation process

Relaxation is a crucial part of the Bellman-Ford algorithm. That escalates the accuracy of the distance to any given vertex. The relaxation operation lasts until the measured distance reduced between vertices comparing that distance with other known distances. The pseudocode for relaxation process:

![Algorithm 2](image)

Figure 2. The pseudocode for relaxation operation

Need to mention that every vertex starts at infinity except the source vertex so it is the starting point for the relaxation. Suppose given edge coming out of the source vertex “A” to another vertex “B”. And also the cost of the edge equal to 4. The “if” condition in the relaxation procedure will be like this for the edge \((A,B)\):

\[
\text{if } B\text{.distance} > A\text{.distance + weight}(B,A)
\]

For sure, this statement is the same as in the below one:

\[
\text{if } \infty > 4.
\]

Thus, if this statement is considered true and the remaining part of the function is executed and \(B\text{.distance} \) is appointed to 4. The predecessor of “B” is set to “A”, the source vertex.

Because of “triangle inequality” its rather secure to operate the relaxation. Another way of interpreting this:

\[
\text{distance}(A,C) \leq \text{distance}(A,B) + \text{distance}(B,C).
\]

Negative cycles detection

A little modification to the Bellman-Ford algorithm allows detecting the negative cycles. It is very necessary because it prohibits the shortest-path finding altogether. The second loop determines the negative edges[7]. The pseudocode is given below:
Numba compiler

Numba is an open source JIT compiler that translates a subset of Python code into fast machine code. JIT compiler based on LLVM compiler. LLVM compiler architecture is given below [8]:

There is no need to replace the Python interpreter or install C++ compiler. Adding specific Numba decorators will accelerate Python functions. While running the code, the decorator added function will be compiled as same speed as machine code. Numba is frequently used in scientific calculations [9].

The results of the experiments

Experimental Setup

Ubuntu 18.04.3 is used as a platform for the implementation of the Bellman-Ford algorithm. The system capacity is Intel® Core™ i5-8250U CPU @ 1.60GHz × 8 and the graphic card is Intel® UHD Graphics 620 (Kabylake GT2). In the below figure, the difference between the Bellman-Ford algorithm in the sequential and parallel cases depicted. The experiment proceeded in several graphs. The very first data contains 170 nodes. Sequential code took 983 milliseconds to run whilst parallel code took 408 milliseconds.

In the second experiment, the graph which has 600 nodes was used. It took 0.636 seconds for parallelized code while sequential one took 5.64 seconds to execute.

In the third experiment, the graph which has 1000 nodes was used. It took 0.937 seconds for parallelized code while sequential one took 8.86 seconds to execute.
According to the results from the plots, it can be seen that parallelized code overcomes sequential one in any case. The interesting fact is, after 1000 nodes the ratio of the computation time does not differ too much.

The experiment proves that applying parallelization in this algorithm is very efficient and reduces the running time of the algorithm in a considerable way.

References


