Parallel implementation the Bellman-Ford algorithm

Mihail Stamenov
Master's Degree in Artificial Intelligence, University of Bologna
mihail.stamenov@studio.unibo.it

Abstract

The goal of this project is to implement the Bellman-Ford algorithm in both C/OpenMP and CUDA C, allowing for performance measurement and comparison. We will describe the parallel implementations, evaluate their performance, and calculate the efficiency of different runs. The implementations will be tested on a small set of graphs, and the results will be saved to files.

1 Introduction

Shortest path algorithms are fundamental in various domains such as routing, network optimization, and artificial intelligence (AI). The Bellman-Ford algorithm is designed for finding the shortest paths between a singled-out source node and the other nodes of the graph. Although Bellman-Ford algorithm scales worse than other algorithms like Dijkstra's algorithm with time complexity of **O(|E|+|V|ln(|V|))** for solving this problem and **O(|V||E|)** for Bellman-Ford, its distinguishing feature is the applicability to graphs with arbitrary weights including negative one.

The Bellman-Ford algorithm has a considerable parallelization resource. The search for shortest paths can be independently performed for all the nodes. In other words we can search for shortest path with many source nodes. On the other hand, the search for shortest paths beginning in a fixed node **u** can also be made in parallel: the initialization of the original paths requires **|V|** parallel operations, and the relaxation of all arcs costs **O(|E|)** parallel operations. Thus, if **O(|E|)** processors are available, then the algorithm terminates after at most **|V|** steps. If the implementation includes check for change on every step the algorithm will stop even earlier - on step ***r*** where ***r*** is the maximum length of the shortest paths outgoing from the chosen source node **u** making the number of needed steps **O(r)**. Of course, relaxing all edges simultaneously is very difficult task because there are many edges with the same destination node means that the distance to this node should be updated at the same time which may cause problems with the writing in the memory if it is not made only from one thread. In this report destination node of edge or arc is the head of the arc. An arc (x, y) is considered to be directed from x to y; y is called the head and x is called the tail of the arc. Moreover, the tail of an arc is called source of an edge in the submitted code and the accompanying report.

As cloted in [] The algorithm of Δ-stepping (Delta-Stepping) can be regarded as a parallel version of the Bellman-Ford algorithm.

In this project is focused on implementing the Bellman-Ford algorithm to detect negative cycles and compute shortest paths in graphs from single source node, making parallel the search of the distances. The project involved creating a set of graphs with and without negative cycles and developing two OpenMP-based solutions and one CUDA-based solution to efficiently compute shortest paths on these graphs. The OpenMP solutions utilize a shared memory model for parallelism, while the CUDA solution leverages GPU parallelism. We aim to assess and compare these implementations in terms of Weak Scaling Efficiency for OpenMP and Throughput for CUDA. These measures will help us understand how efficiently each solution scales with increasing graph sizes and thread counts.

2 System description

The project aimed to solve the shortest path problem using both OpenMP and CUDA parallel processing. Two OpenMP solutions were developed, each with a unique graph representation and work distribution strategy:

Destination-based Solution (dest): Each node in this representation holds a list of all incoming

neighbors pointing to it. The algorithm assigns each thread a node, and the thread iterates through the incoming neighbors to check if it can update the shortest distance. If an update is possible, the thread performs the operation. The process is repeated until no further updates occur, indicating that the shortest paths are fully computed. If updates continue during the last iteration, the algorithm detects a negative cycle.

Source-based Solution (source): In this representation, each node stores a list of its neighbors (outgoing edges). The algorithm distributes the neighbors of a node among the available threads, where each thread works on one neighbor until completion. The process continues for all nodes. Like the dest solution, the algorithm stops when no further updates are detected.

CUDA-based Solution: The CUDA implementation is conceptually similar to the source solution in OpenMP, but leverages the massive parallelism offered by GPUs. Threads are assigned to process neighbors of a node, and updates occur simultaneously across multiple threads for efficient path computation.

Both OpenMP and CUDA solutions track whether changes occurred during each iteration. If no changes are detected, the algorithm halts. If changes continue during the final iteration, it indicates the presence of a negative cycle.

3 Experimental setup and results

The solutions were tested on 18 different graphs, divided into two categories: 9 graphs without negative cycles and 9 graphs with negative cycles. These graphs vary in the number of nodes and maximum neighbors per node, ranging from 512 nodes with 512 neighbors to 8192 nodes with 8192 neighbors. The graphs were generated to avoid multigraphs, as only the smallest edge weight between two nodes is considered for shortest path calculations.

The table below summarizes the graph properties and their respective edge counts:

The experiments measured the performance of both OpenMP solutions (dest and source) and the CUDA solution in terms of execution time, efficiency in detecting negative cycles, and scalability across different graph sizes.

Number of Nodes	Max Neighbors	Number of Edges Has
512	512	133,084
512	512	136,784
1024	512	265,113
1024	512	263,866
1024	1024	530,673
1024	1024	539,496
2048	1024	1,056,20
2048	1024	1,066,40
2048	2048	2,094,84
2048	2048	2,085,53
4096	2048	4,181,07
4096	2048	4,188,26
4096	4096	8,315,29
4096	4096	8,577,26
8192	4096	16,673,97
8192	4096	16,780,66
8192	8192	33,368,60
8192	8192	33,248,45

Table 1: Graph properties and number of threads used for each experiment

4 Discussion

The results show that the best performing model is Model CPS, scoring 0.735 of F1-score on validation set, safely above our baselines scoring 0.432 and 0.526. From table **??** we can see an important skew in how our models perform over each class, in particular Model C classify almost all entries to Conservation and Self-transcendence, as a majority classifier would do. Our best model, Model CPS, instead can correctly classify a larger number of entries with more balance between the classes. Setting dropout probability to 0.3 in conjunction with the cosine with restarts scheduler resulted in the best performances over all the models, confirming that greater generalization helps in tasks where the class distributions are very skewed, such in this case.

5 Conclusion

Human value detection from text is task that is still an open problem. Our model perform well over the dataset, scoring 0.735 F1-score with more resilient performances over less frequent classes. BERT and other transformers models have been proved to work quite well in extracting semantic meaning from text. One limitation found is that our models are prone to over-fitting even with the dropout regularization.

References