Parallel Bellman-Ford: Implementation and Analysis

APAI Project 2023/24

Maria José Valente da Silva Carneiro - 1900113521

Alma Mater Studiorum - University of Bologna
maria.josecarneiro@studio.unibo.it

Abstract. This work investigates the parallel implementation and analysis of the Bellman-Ford algorithm, a crucial method for finding the shortest path in weighted graphs. Two parallelization strategies are explored: leveraging CPU threading with OpenMP and utilizing GPU parallel architecture through CUDA. Experiments on graphs of varying sizes and edge weights demonstrate the effectiveness and scalability of both implementations, with CUDA outperforming OpenMP due to its enhanced threading capabilities. Overall, the study underscores the potential of parallelization in accelerating shortest-path computations.

Keywords: Bellman-Ford, Parallel Programming, OpenMP, CUDA

1 Introduction

The Bellman-Ford algorithm is a fundamental tool in graph theory and network optimization, offering a robust solution for finding the shortest path in a graph with weighted edges. This algorithm has found extensive applications in various domains, such as network routing, traffic planning, and compiler optimization.

The essence of the Bellman-Ford algorithm lies in its ability to handle graphs with both positive and negative edge weights, making it a versatile choice in scenarios where other algorithms, like Dijkstra's, fail due to the presence of negative edges. By iteratively relaxing edges, the algorithm gradually improves its estimates of the shortest paths until convergence, effectively providing the shortest path from a single source vertex to all other vertices in the graph. While the sequential Bellman-Ford algorithm demonstrates its efficacy in solving single-source shortest-path problems, its inherently iterative nature presents opportunities for parallelization. With the advent of multi-core processors and massively parallel computing architectures like GPUs, exploring parallel implementations of the Bellman-Ford algorithm is both relevant and promising.

This work explores the implementation and analysis of parallel versions of the Bellman-Ford algorithm, focusing on two distinct parallelization strategies: one leverages the threading capabilities of CPUs using OpenMP and the other exploits the parallel architecture of GPUs through CUDA. The goal is to explore the effectiveness of parallelization in accelerating the computation of shortest paths, comparing the performance of these implementations against their single-threaded version, and examining their scalability with increasing problem sizes.

This work is organized as follows: Section 2 describes the sequential Bellman-Ford algorithm, Section 3 briefly presents the tools explored in this work, Section 4 details the proposed parallel implementations, Section 6 analyzes the results obtained from experiments and 7 offers some concluding remarks.

2 The Bellman-Ford Algorithm

Given a directed or undirected graph $G(V, E)$ where V represents the set of vertices and E the set of edges, the Bellman-Ford algorithm finds the distances that define the shortest path between a starting vertice s and the rest of the vertices in G, given that the graph has no negative cycles [1]. A negative cycle can be defined as a cycle in the graph where the distance perpetually decreases since another iteration yields a shorter path than the last. The sequential algorithm can be defined as follows.

Algorithm 1 Bellman-Ford's Algorithm

```
for vertex $u \in G(V)$ do
    $dist(u) \leftarrow \infty$
end for
$d(s) = 0$
for $i$ from 1 to $|V| - 1$ do
    for edges $(u, v) \in G(E)$ do
        if $d(u) + weight(u, v) < d(v)$ then
            $d(v) = d(u) + weight(u, v)$
        end if
    end for
end for
for edges $(u, v) \in G(E)$ do
    if $d(u) + weight(u, v) < d(v)$ then
        $G$ has a negative cycle
    end if
end for
```

The algorithm iteratively relaxes the edges as the distances are updated until convergence and has a time complexity of $O(|V||E|)$ and a space complexity of $O(|V|)$.

3 Libraries and Tools

The project explores parallel implementations with the help of the OpenMP and CUDA tools.

OpenMP [2] is an API specification for parallel programming exploited by the compiler, responsible for handling thread management. In source code, `#pragma` statements allow code regions to be implemented in parallel, given that the program is compiled using the flag `-fopenmp`. In the project, OpenMP version 10.2.1 was used, bundled with g++-12.

CUDA [3] is an API specification for parallel programming created by NVIDIA, allowing developers to build parallel applications that run directly on NVIDIA's GPUs, with significant speedups compared to CPU parallelized approaches. CUDA can be used only with machines that are compatible with NVIDIA GPUs. In the project, CUDA version 11.4 was used.

4 Parallel Implementations

The complexity of the Bellman-Ford algorithm, both for time and memory management, is intrinsically linked with the number of relaxation operations it performs, as they are the most expensive. Both approaches aimed to target this operation and parallelize it to diminish the workload on a single thread.

Careful allocation of dynamic memory and storing graph data in 1D arrays instead of 2D was applied in both implementations for adequate memory management and faster data access.

Another optimization applied to both versions consisted of early stopping when there was no change in distance values in the previous iteration since the result would remain the same, reducing the runtime to the depth of the shortest path tree in G rooted in s. This was done by keeping a flag for each thread that would be positive once there was any change in distances: if all threads didn't report any change, the calculation could be halted as the algorithm already had computed the right results.

The general idea for both approaches was to divide the workload between threads so that each thread handled only a subset of vertices with no overlap with other threads. This was done to avoid concurrency issues without the overhead of using pragma statements for critical sections or atomic instructions, for the OpenMP case at least. To do so, only the inner loop that traversed graph edges was parallelized and divided among threads. All threads updated the distance array with no concurrency issue since each thread handled separate vertices and updated the distance accordingly. Negative cycles were detected by performing one more iteration and verifying if the distances had changed. This check and the early convergence check had to be done by a single thread and could not be parallelized.

In the OpenMP implementation, threads were divided according to the number of vertices: if the number of vertices was less than the number of threads, some threads would be idle and have to wait for the others to finish; if it was higher, vertices were equally distributed by threads, with the chance some could get slight more vertices to work on. Barriers were applied to ensure threads waited for each other before performing checks and moving on to the next iteration.

In the CUDA implementation, the block structure of the GPU was exploited to divide work among threads while still dividing vertices by threads. The number of blocks per grid was calculated according to the number of threads in each block to handle arbitrary graph vector sizes.

5 Experimental Setup

The results were obtained by executing the OpenMP and CUDA implementations of the parallel Bellman-Ford algorithm in a machine with an Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz with 4 cores and Turing GPUs driven with NVIDIA v. 470 drivers with 512 threads. The experiments performed for testing were done on the following graphs:

– 1000, 2000, 3000, and 4000 vertices with 0.8 edge probability positive weight graphs
– 2000 vertices and 0.8 edge probability negative and positive weight graph (contain negative cycles)

OpenMP tests were performed for 1, 2, 4, and 8 threads, the last of which is a virtual case, as there are only 4 cores in the machine. The CUDA tests were done on multiple of 32 block dimensions of 32, 64, 128, and 256 as the warp size of the NVIDIA GPU is 32. This helped evaluate the scalability and resource management of both parallel implementations. Algorithm results were written to files and posteriorly compared with the corresponding results of a sequential implementation to verify correctness. All test cases provided correct distance computations and identification of negative cycles.

6 Results and Analysis

The results in Table 1 refer to experiments run on graphs with increasing vertices with positively directed edges, with OpenMP. The computation of the best path between nodes quickly converges and computation finishes relatively early when comparing with a version without the early convergence check. The best runtime was obtained using 4 threads for all cases, which makes sense considering the number of physical cores in the machine. Parallelizing the algorithm using OpenMP yielded an almost 3 times faster computation than a single-threaded version. Considering larger graphs and scalability, the speedup and efficiency remain similar, which means the algorithm is robust to data increase. However, the execution time increases significantly with the number of vertices due to the equivalent increase in the number of barriers, as the inherent structure of the algorithm after each edge traversal requires high synchronization.

As for CUDA, the results in the Table of Fig 1 refer to experiments run on the same graphs used for the OpenMP experiment. Generally, the execution time is lower for all numbers of vertices compared to the OpenMP version. Considering the block dimensions, since the number of blocks per grid is computed according to the pre-defined number of threads per block (equal to the block dimensions, so there is one thread per vertex), there is almost no difference between using any of the block dimensions, as they dynamically adapt to accommodate changeable arrays.

Directly calculating the speedup with respect to the best-performing version of the OpenMP implementation (4 threads) and for 2000 vertices, for example, we get a speedup of 5.12, meaning it is more attractive to use CUDA for parallelization as

Vertices	**1000**			**2000**			**3000**			**4000**		
Threads	Time (ms)	Speedup	Strong Efficiency	Time (ms)	Speedup	Strong Efficiency	Time (ms)	Speedup	Strong Efficiency	Time (ms)	Speedup	Strong Efficiency
1	19.77	1.0	1.0	79.22	1.0	1.0	151.41	1.0	1.0	272.56	1.0	1.0
2	14.93	1.32	0.66	41.06	1.93	**0.96**	77.83	1.95	**0.97**	143.63	1.90	**0.95**
4	6.67	**2.96**	**0.74**	33.53	**2.36**	0.59	55.83	**2.71**	0.68	97.85	**2.79**	0.70
8	11.85	1.67	0.21	35.34	2.24	0.28	72.04	2.10	0.26	106.27	2.57	0.32

Table 1: Results for a positive directed graph using the OpenMP parallel implementation of the Belman-Ford algorithm.

it effectively processes the calculations faster. This is due to the fact that there are more threads to work within the GPU in comparison with the number of threads in the CPU, which is beneficial, especially for matrix-on-matrix computations. The direct comparison between execution time given the number of vertices is present on the graph of Fig 1.

Vertices	**1000**	**2000**	**3000**	**4000**
Block Size	Time (ms)	Time (ms)	Time (ms)	Time (ms)
32	2.31	6.53	8.40	11.36
64	2.31	6.55	6.98	11.29
128	2.32	6.57	8.41	11.31
256	2.33	6.64	8.49	11.40

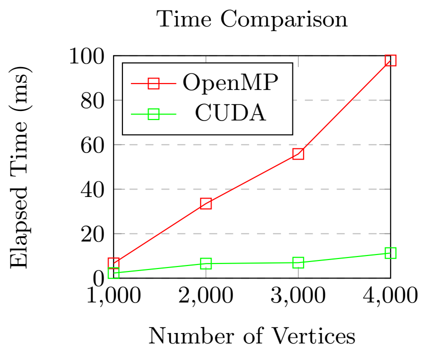

Fig. 1: Comparison of results for a positive directed graph using the CUDA parallel implementation of the Belman-Ford algorithm with the OpenMP implementation.

The last experiment considered graphs with negative weights that contain negative cycles. In this case, the execution time was significantly higher for the OMP version for all threads. This is due to the extra iteration needed to verify if the algorithm keeps giving decreased distances since the algorithm never breaks early as it never converges, so complexity increases to $O(V^3)$. As an example, table 2 has the execution time of the OpenMP and CUDA implementations for a graph with 2000 vertices that contains a negative cycle. The CUDA version was executed with a block size of 64 and the OpenMP was executed with 4 parallel threads, as they were the best-performing versions for each implementation. As observed, the CUDA version significantly outperformed the OpenMP version due to its increased threading potential that more efficiently parallelizes the algorithm, with an impressive speedup of almost 10 concerning the OpenMP implementation.

	Time (ms)	Speedup
OpenMP	14631.82	1.0
CUDA	1481.54	9.876

Table 2: Results with OpenMP and CUDA for a directed graph with 2000 vertices and negative weights.

Generally, the CUDA implementation outperformed the OpenMP implementation as it directly exploits the GPU functionalities, providing the best possible environment for parallel execution. However, given the CPU limitations, the OpenMP version also showed promising results, especially when comparing among threads.

7 Conclusions

In this work, the Bellman-Ford algorithm and its capacity for parallelization were explored and implemented while testing the impact of software and hardware in parallel settings. The primary objective was to guarantee the correctness of the implementation and avoid race conditions while trying to optimize thread distribution and overall execution. This was achieved as seen by the results where the GPU-oriented approach with CUDA performed better, while the CPU-oriented version lacked when handling increased operations. However, the high synchronicity required by the inherent structure of the algorithm was a handicap in achieving better parallelism. Other approaches that aim to bridge this gap could be explored in future work, considering a better redesign of the simple algorithm, as well as testing with better and more powerful machines.

References

1. Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs.* Springer London, 2009.
2. OpenMP Architecture Review Board. Openmp architecture review boardopenmp application programming interface version 4.5, 2015.
3. NVIDIA Corporation. Cuda documentation, 2021.