Parallel implementations of the Bellman-Ford algorithm for OpenMP platforms and CUDA architectures

Kilian Tiziano Le Creurer

* *E-Mail: kilian.lecreurer@studio.unibo.it*

Abstract

The study focuses on implementing and evaluating multiple parallel versions of the Bellman-Ford algorithm, renowned for solving single-source shortest paths in graphs. The research evaluates the different implementation performances across various setups to asses their scalability. Two C extensions, OpenMP and CUDA, are employed to implement the algorithm, allowing for parallel processing in different architectures. The results obtained during this study were leveraged to grasp some insights into the effectiveness and limitations of parallel architectures and plaforms

Keywords: Bellman-Ford ; parallel programming ; OpenMp ; Cuda ; C

Contents

1. Introduction

The Bellman-Ford algorithm [4] [5] is a fundamental method used in graph theory and network analysis. It addresses the single-source shortest path problem in weighted graphs, aiming to determine the shortest path from a source vertex to all other vertices. Formally, this can be represented as finding the shortest distances $d[v]$ from a source vertex s to all other vertices $v \in V$ in a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges (represented as a triple (u, v, w)). The Bellman-Ford algorithm solves the SSSP problem by iteratively relaxing all the edges of the graph. This research focused on the adaptation of several versions of the algorithm to run efficiently on parallel architectures and platforms. The aim was to understand the effectiveness and scalability of this algorithm on parallel architectures. Two sequential implementations of the algorithm were adapted to run in parallel architectures by different means. Specifically, I propose three OpenMP versions, namely *omp-simple*, *omp-locks*, and *omp-frontier*, and two CUDA versions, namely *cuda-simple* and *cuda-frontier*. The evaluation of those implementations showed that when running on a multi-core CPU the algorithm scales well up to the physical number of cores when the synchronization is maintained limited. On the other hand, when running on a CUDA device, the algorithm suffered less from this scalability issue.

Section 2 reports a small discussion on the parallel hardware architectures, Section 3 contains the descriptions of the sequential versions of the Bellman-Ford algorithm, Section 4 details the main ideas behind the algorithm parallelization, Sections 5 and 6 contain the experimental setup and results, and in section 7 there's a brief recap.

2. Parallel architectures and platforms

Parallel hardware has become widespread, representing the easiest means to boost computing power without altering transistor size. Yet, crafting parallel software isn't simple. This challenge has motivated numerous companies and research groups to devise standards for constructing and implementing parallel software. OpenMP [3] furnishes a framework for shared memory parallel computation; its adaptability across diverse architectures solidifies its status as a robust standard. Conversely, CUDA [1] serves as a specialized language for NVIDIA GPGPUs, establishing itself as a standard in various domains.

3. Bellman-Ford

Given a weighted graph $G = (V, E)$, where each edge $e \in \mathbb{E}$ is a triple $e = (u, v, w)$ representing the two nodes and the weight, the sequential version of the Bellman-Ford algorithm consists of iterating $V - 1$ times a relaxation procedure over all the edges of the graph. Then a last relaxation is performed to understand whether negative cycles in the graph exist.

Algorithm 1 Bellman-Ford sequential

```
for all $v \in \mathbb{V}$ do
    $dist[v] = \inf$
end for
$dist[src] = 0$
$negative = False$

for $i \leftarrow 0$ to $V - 1$ do
    for all $(u, v, w) \in \mathbb{E}$ do
        if $dist[u] + w < dist[v]$ then
            $dist[v] = dist[u] + w$                    ▷ Relaxation procedure
        end if
    end for
end for

for $i \leftarrow 0$ to $V - 1$ do
    for all $(u, v, w) \in \mathbb{E}$ do
        if $dist[u] + w < dist[v]$ then
            $negative = True$                          ▷ Negative cycle detected
        end if
    end for
end for
```

The algorithm's time complexity of $O(V \cdot E)$, where V is the number of vertices and E is the number of edges, poses a drawback. This complexity becomes inefficient for larger graphs or denser networks w.r.t. other solutions, such as Dijkstra [6]. This version of the algorithm can be improved for some specific cases, by introducing a breaking procedure. The algorithm can indeed terminate if a relaxation step does not produce any update on the $dist$ array.

A refined version of the Bellman-Ford algorithm is been also implemented during the research. The idea is been taken from [2] and is based on the utilization of a couple of frontier arrays, to keep track and relax the minimum amount of edges at each step.

Algorithm 2 Bellman-Ford frontier based

```
for $\forall$ vertices $u \in V(G)$ do $d(u) = 1$
end for
$d(s) = 0$
$F1 = \{s\}$
$F2 = \{\}$
while $F1 \neq \varnothing$ do
    for all $u \in F1$ do
        $u$ = DEQUEUE($F1$)
        for all $v \in adj[u]$ do
            if $d(u) + w < d(v)$ then
                $d(v) = d(u) + w$
                ENQUEUE($F2, v$)
            end if
        end for
    end for
SWAP($F1, F2$)
end while
```

The idea is simple and powerful, at each relaxation step if the distance to a certain node is updated, then that node goes in the frontier for the successive relaxation. This way it is possible to avoid useless relaxation checks.

4. Parallel Implementations

The adaptation of the above sequential versions to parallel hardware has been performed by splitting each relaxation procedure across multiple cores. Every relaxation step consists of reading data and eventually updating the distances array $dist$. The distances array must be accessed at a certain position by only one thread at a time. This behavior is been ensured in two ways.

- The parallelization is limited to the inner loop. In this way, it is ensured that each thread has its portion of the distances array to access.
- Ensure that when a thread accesses a distance, all the other ones must wait until it has finished. This is been implemented only in the OpenMP version through the use of locks, to avoid serialization.

5. Experimental setup

All the experiments were run on the GPU-equipped HPC (High Performance Computing) cluster of DISI unibo. The OpenMP experiments were performed on a Quad-Core Intel(R) i9-9820X and a Quad-Core Intel (R) Xeon(R) W-2123, while the CUDA ones were performed on 512 threads NVIDIA Turing GPUs. The experimental pipeline is the following: for each type of generated graph a baseline sequential version of the algorithm is run to obtain the correct distances (the baseline is a simple version of the Bellman-Ford algorithm), and then all the OpenMP and CUDA implementations are run through and their results are checked through a direct comparison between files. The experiments performed are the following:

- 500 and 1000 nodes fully connected graphs without negative cycles.
- 500 and 1000 nodes fully connected graphs with negative cycles.
- 3000 nodes graph with negative cycle.

Regarding OpenMP, the parallelization schedule for the loops is chosen to be static, because the amount of job for every thread is more or less the same. Regarding CUDA, memory preallocation is been adopted to improve performances over a loop of multiple graphs.

6. Results

The first results are on graphs without negative cycles, this characteristic makes the algorithm generally faster when running on fully connected graphs because after a few relaxation steps, the algorithm finds the best distances and breaks. This makes the first loop shorter, and consequently the amount of synchronization between threads smaller.

Threads	Time (ms)	Speedup	Strong Efficiency
1	3.6	1.0	1.0
2	2.0	1.8	**0.9**
4	1.2	**3.0**	0.75
8 (Virtual)	2.3	1.6	0.2

Fig. 1: OMP-Frontier results on an Intel(R) i9-9820X when run on positive weighted graphs.

Threads	Time (ms)	Speedup	Strong Efficiency
1	3.83	1	1
2	2.90	1.32	0.65
4	2.05	1.86	0.46
8 (Virtual)	3.09	1.23	0.15

Fig. 2: OMP-Frontier results on an Intel(R) Xeon(R) W-2123 when run on positive weighted graphs.

Threads	Time (ms)	Speedup	Strong Efficiency
1	5.6	1.0	1.0
2	3.5	1.6	0.80
4	2.1	2.7	0.67
8	4.5	1.2	0.15

Fig. 3: OMP-Simple results on an Intel(R) i9-9820X when run on positive weighted graphs.

Threads	Time (ms)	Speedup	Strong Efficiency
1	6.45	1	1
2	4.06	1.59	0.80
4	3.28	1.96	0.49
8 (Virtual)	5.54	1.16	0.15

Fig. 4: OMP-Simple results on an Intel(R) Xeon(R) W-2123 when run on positive weighted graphs.

Threads	Time (ms)	Speedup	Strong Efficiency
1	5.1	1.0	1.0
2	3.2	1.6	0.8
4	2.1	2.4	0.6
8 (Virtual)	3.6	1.4	0.2

Fig. 5: OMP-Locks results on an Intel(R) i9-9820X when run on positive weighted graphs.

Threads	Time (ms)	Speedup	Strong Efficiency
1	5.98	1	1
2	6.27	0.95	0.48
4	3.50	1.70	0.42
8 (Virtual)	4.7	1.26	0.16

Fig. 6: OMP-Locks results on an Intel(R) Xeon(R) W-2123 when run on positive weighted graphs.

Fig. 7: OpenMP results on graphs without negative cycles, the results is the average between size 500 and size 100

The results show that the frontier approach is better, both in terms of sequential wall time (one thread) and scalability. In all implementations, the speedup is limited, especially when running on the Xeon processor. This is caused by the structure of the algorithm, which requires a high synchronization between threads, when the size of the graphs increases the number of barriers grows accordingly, introducing delay. This behavior is highlighted when the algorithm is run on negatively weighted graphs, where the number of barriers becomes equal to the number of nodes v in the graph. The CUDA approaches are evaluated by computing the wall time and the speedup w.r.t. to the OpenMP corresponding versions. Here the speedup is very limited, possibly because synchronizing a high number of threads so many times has a notable computational cost. The observation of the results over several runs showed that the frontier-based approach written in CUDA may be slower due to the dynamic allocation of shared memory at every iteration. Even if the CUDA device is highly parallelized, in this setup the advantage w.r.t. is not as significant. This is because the Bellman-Ford algorithm needs several synchronization steps, which may become inefficient when running with a high number of threads.

	Wall time (ms)	**SpeedUp w.r.t OMP**
CUDA-frontier	1.40	2.73
CUDA-simple	0.94	6.84

Table 1: Wall time and Speedup on positive weighted graphs. The speedup is computed w.r.t. the corresponding OMP model with one thread

The second experiment reported considers graphs with negative cycles. In this setup, the complexity is fixed to $O(v^3)$ where v is the number of vertexes in the graph, so the throughput for CUDA implementation is also reported. The performance of the OpenMP versions worsens significantly, especially the scalability of the OMP-Locks implementation. In particular, only the frontier-approach solution keeps being scalable. Interestingly, the wall time with 8 threads is often less than the case with 4, which is the physical number of threads.

Threads	**Time (s)**
1	3.18
2	1.94
4	1.63
8	1.44

(a) OMP-Frontier approach on negative weighted fully-connected graphs on Intel (R) Xeon(R) W-2123

Threads	**Time (s)**
1	2.70
2	5.93
4	6.13
8	2.70

(b) OMP-Simple approach on negative weighted fully-connected graphs on Intel (R) Xeon(R) W-2123

Threads	**Time (s)**
1	8.9
2	37.2
4	20.7
8	21.1

(c) OMP-Locks approach on negative weighted fully-connected graphs on Intel (R) Xeon(R) W-2123

Fig. 8: OpenMP results on graphs with negative cycles, the results is the average between size 500 and size 100

The setup with negative weighted graphs is where the CUDA version shows a significant advantage w.r.t. the corresponding sequential algorithm. This is mostly true for the Simple version of the algorithm.

	Time (s)	**SpeedUp**	**Throughput (M/s)**
CUDA-simple	0.061	18.45	2041
CUDA-fronter	0.095	5.37	1309

(a) 500 nodes graphs

	Time (s)	**SpeedUp w.r.t OMP**	**Throughput (M/s)**
CUDA-simple	0.33	23.83	3055
CUDA-fronter	0.54	6.56	1832

(b) 1000 nodes graphs

Table 2: Wall time, Speedup, and Throughput (M ops/sec) on negative weighted graphs. The speedup is computed w.r.t. the corresponding OMP model with one thread

The last experiment consisted of comparing CUDA and OpenMP algorithm versions using a single graph with 3000 nodes.

	Wall time (s)	**SpeedUp w.r.t OMP**	**Throughput (M/sec)**
CUDA-simple	18	9.48	3350
CUDA-frontier	13	8.18	1500

Table 3: Performance metrics for a 3000 vertexes graph.

In all the presented experiments, the Frontier-Based method proved to be worse than the Simple version of the algorithm. This behavior is thought to be due to the increased amount of shared memory needed every time the kernel is run.

7. Concluding remarks

This project explored the world of parallel architectures and platforms, by working directly hands-on on a practical application, the Bellman-Ford algorithm. The experiments showed that parallelization is suitable for the Bellman-Ford algorithm, even though its scalability when the size of the graph or when the number of threads increases is limited, due to its intrinsic sequential nature, which requires synchronization barriers. The experiments produced during this project made clear both why parallel programming became ubiquitous and why it is not straightforward to exploit its features in practical challenges.

Some possible improvements of this research could include the introduction of new versions of the algorithm, both sequential and parallel, and the evaluation on CPU devices with a higher number of physical cores, to increase the scalability analysis.

References

[1] Jayshree Ghorpade et al. "GPGPU Processing in CUDA Architecture". In: *Advanced Computing: An International Journal (ACIJ), Vol.3, No.1* (2012).

[2] Federico Busato and Nicola Bombieri. "An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures". In: *IEEE Transactions on Parallel and Distributed Systems* 27.8 (2016), pp. 2222–2233. DOI: 10.1109/TPDS.2015.2485994.

[3] Dagum L. Menon N. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* (1998).

[4] Bellman R. *On a routing problem*. Vol. 16. 1. SIAM journal, 1958, pp. 87–90.

[5] Ford L. R. *Network Flow Theory*. 1956.

[6] Dijkstra E. W. "A note on two problems in connexion with graphs". In: *Numerische Math* 1.1 (1959), pp. 279–281.