Report Parallel Bellman-Ford

alessio pellegrino (alessio.pellegrino@studio.unibo.it)

July 20, 2024

1 Introduction

The Bellman-Ford algorithm is a well-known technique for graph exploration, particularly for finding the shortest path from a given source node to every other node in the graph. The time complexity of the algorithm is $O(|V| \cdot |E|)$, where $|V|$ represents the number of vertices in the graph and $|E|$ denotes the number of edges. Although Bellman-Ford is slower than Dijkstra's algorithm, it is more versatile as it can handle edges with negative cycles.

In this report, we will explore two different implementations of the Bellman-Ford algorithm, with a focus on parallelizing its execution. Additionally, these implementations will be realized using both OpenMP (OMP) and CUDA frameworks.

2 Bellman-Ford parallelization

The Bellman-Ford algorithm offers four potential points for parallelization: (i) Finding an *infinite* value, (ii) Initialization of the distances and predecessors array, (iii) The relaxation procedure, and (iv) The negative cycle check.

1. Finding an Infinite Value: To determine an *infinite* value, we need to examine all the edges of the graph, find the maximum edge weight, and add one to it to ensure it is the highest possible value found in the graph. This step can be parallelized using the reduction paradigm to find the maximum value among the edges. This approach is necessary because an actual infinite value cannot be represented in real-world scenarios, requiring us to find a suitable alternative.

2. Initialization of Arrays: Initializing the distance and predecessor arrays does not require a specific parallelization paradigm since each operation in the for loop is inherently independent. Therefore, we can simply parallelize the loop to initialize these arrays.

3. The Relaxation Procedure: The relaxation procedure requires a more detailed analysis. At first glance, it seems possible to parallelize the entire for loop over all the edges. However, doing so would result in inconsistencies in the computation of the distances. To resolve this, we can parallelize the assignment of the distance for each node. This way, we fix the distances of all other nodes, and the procedure can be seen as finding the minimum of a sequence where the elements are the distances of all other nodes plus the weight of the corresponding edge. This can be effectively solved using the reduction paradigm. The pseudocode for the parallel relaxation is shown in Algorithm 1.

Algorithm 1 Parallel relaxation code

```
1: for i = 1 to |V[G]| − 1 do
2:     for each node u in |V[G]| do
3:         parallel do
4:             for each edge (u′, v) ∈ E[G] where u = u′ do
5:                 RELAX(u, v, w, d, π)
6:             end for
7:         end parallel
8:     end for
9: end for
```

4. Negative Cycle Check: The final search for a negative cycle can be easily parallelized by assigning a thread to each iteration of the loop and utilizing the reduction paradigm.

2.1 Implementations

name	nodes	edges	name	nodes	edges
graph 0	50	498	graph 9	500	188,430
graph 1	50	1,948	graph 10	3,000	49,926
graph 2	100	1,994	graph 11	3,000	49,921
graph 3	100	7,706	graph 12	3,000	49,910
graph 4	200	9,662	graph 13	3,000	49,9033
graph 5	300	47,510	graph 14	5,000	11,982,296
graph 6	300	75,912	graph 15	8,000	31,934,988
graph 7	200	28,480	graph 16	9,000	40,417,767
graph 8	500	95,205	graph 17	10	50

Table 1: Input graphs and their characteristics

For this project, I propose two possible implementations:

1. Adjacency Matrix Implementation (nodes): In this implementation, I use an adjacency matrix to store the graph. I also implement a speed-up mechanism: at each iteration, I keep a counter of how many updates are made in the relaxation loop. If no updates occur during a full iteration, I can terminate the cycle early, avoiding the need to complete all $|V[G]|-1$ iterations. Both the OpenMP (OMP) and CUDA versions share this logic, but the adjacency matrix is stored differently. In the OMP version, it is saved in a 2-dimensional array, while in the CUDA version, it is stored in a 1-dimensional array to facilitate data transfer to and from the GPU.

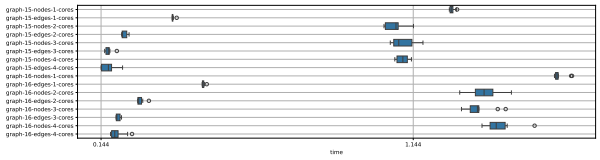

Figure 1: Variation of the execution time for a given combination of input, version, and number of cores.

2. Array of Edges Implementation (edges): In this implementation, I use an array of nodes and implement a different speed-up mechanism: instead of stopping after $|V[G]|-1$ iterations, the algorithm runs for $|V[G]|$ iterations. The last iteration checks for any further changes, and if changes are detected, it indicates a negative cycle in the graph. The CUDA and OMP versions differ slightly due to the varying number of edges for each node.

In the OMP version, I use $|V[G]|$ arrays of varying sizes, each with an integer value indicating its size. In the CUDA version, I maintain a 1-dimensional array and add a secondary array that contains the number of edges for each node. This allows me to access the edges of node $i+1$ by starting from the last edge of node i. The first edge of node $i+1$ will be the next edge in the array, and the number of edges for node $i+1$ can be read from the secondary array.

3 Performance analysis

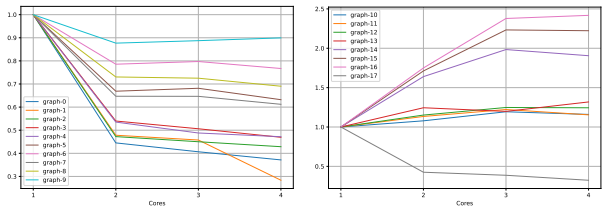

Figure 2: Speedup on the edge version.

In this section, I will analyze the performance of each implementation in both CUDA and OMP. Each implementation has been evaluated using the same set of inputs, whose characteristics are shown in Table 1. To ensure consistency, I repeated each experiment 10 times on two different machines: a cluster and a laptop. Here, I present only the cluster results, while the corresponding graphs for the local laptop are included in the appendix.

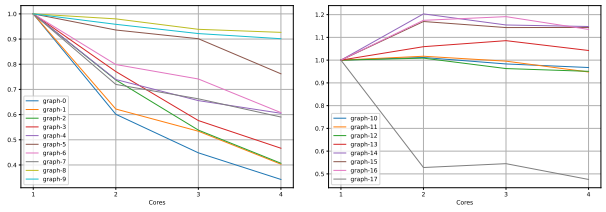

Figure 3: Speedup on the node version.

3.1 OMP

To compute each metric, I ran the experiments with a number of cores varying from 1 to 4. Figure 1 shows the variation of the results over the ten runs for some of the graphs on different cores. Given that the difference in computational time is reasonably small, from now on I will consider only the mean execution time for a given graph on a given number of cores.

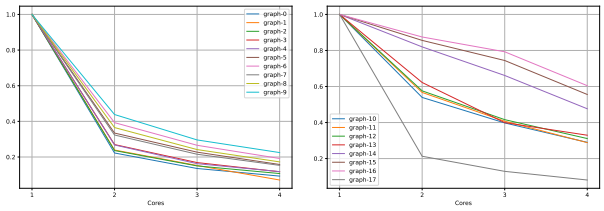

Figure 4: Strong efficiency on the edge version.

Figures 3 and 2 show the speedup of each version as a function of the number of cores. The speedup was computed as $\frac{T(1)}{T(p)}$ where $T(x)$ is the execution time on x cores. Both versions of the algorithm show a performance loss on small inputs (e.g., less than 1k nodes or 50k edges), indicating that the cost of parallelization is higher than the gain of processing the input in parallel. However, there is a significant difference with larger graphs: the edge version is notably faster than the node version, reaching almost a 2.5x speedup compared to the 1.2x of the node implementation. This indicates that the edge version is more suitable for parallel execution. Another indication of better parallelism in the edge version is that the best performance is achieved with 3 cores, while the node version peaks at 2 cores.

Figures 4 and 5 present the strong efficiency for the edge and node implementations, respectively, as a function of the number of cores. The strong efficiency is calculated using the formula $\frac{T(1)}{pT(p)}$, where $T(x)$ represents the execution time on x cores, and p is the number of cores. This metric helps in understanding how well the algorithm scales with increasing computational resources while keeping the problem size constant.

Similar to the speedup results, the edge version of the algorithm achieves superior performance overall, particularly when dealing with larger inputs. This indicates that the edge-based implementation can effectively leverage the additional cores to reduce execution time, maintaining higher efficiency levels even as the number of cores increases. In contrast to the

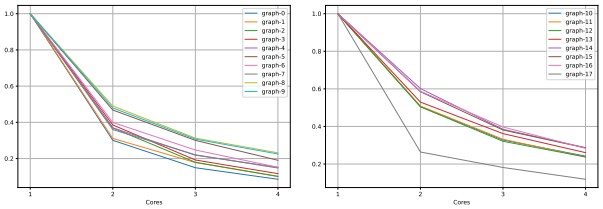

Figure 5: Strong efficiency on the node version.

speedup, the strong efficiency results exhibit a distinct trend. For the edge version, the efficiency remains relatively high with the use of 3 cores, and the efficiency curve is flatter for the three largest input graphs (Graphs 14, 15, and 16). This flatter curve suggests that the edge version is able to sustain its efficiency gains across multiple cores, particularly when processing larger inputs. The flatter curve indicates a more stable performance, highlighting the algorithm's capability to handle larger problems without a significant drop in efficiency as more cores are utilized.

On the other hand, the node version follows a trend similar to that observed with smaller inputs but at a higher efficiency point. While the node version does show some improvement with increased core counts, its efficiency gains are not as pronounced as those seen with the edge version. This suggests that the node-based implementation may have inherent limitations in parallelizing certain operations or managing workload distribution effectively across multiple cores.

Figure 6 shows the weak efficiency on both the edges and nodes versions. The weak efficiency was computed as $\frac{T(1,n)}{T(p,np)}$ where $T(x,k)$ is the execution time of a problem of size k on x cores, and p is the number of cores. To compute the weak efficiency measure, four special graphs were created. These graphs are fully connected with a base number of

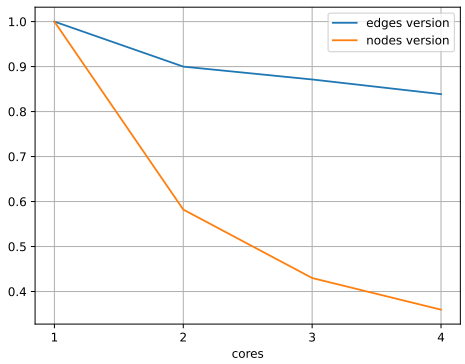

Figure 6: Weak efficiency.

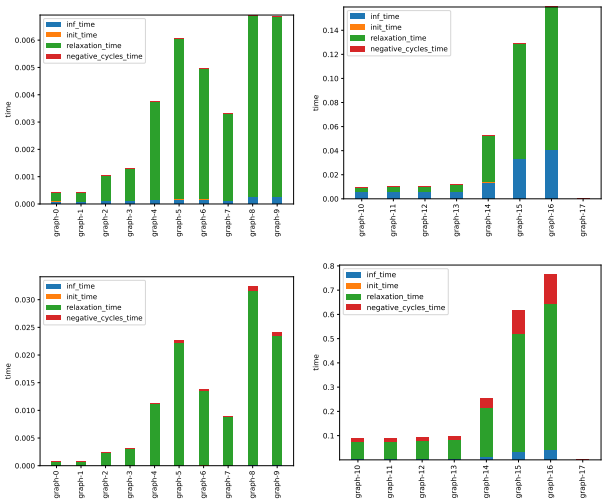

Figure 7: Average time taken by each section of the algorithm on the edge (top) and nodes (bottom) versions on OMP.

nodes of 1000 that scaled linearly with the number of cores (1000 for a single core, 1259 for two cores, 1442 for three cores, and 1587 for four cores). This ensured consistency and a fair metric measurement.

The results show that the edge version scales much better than the node version, always staying above 0.8, which denotes very good scalability. On the other hand, the node version scales poorly, with much worse results that continue to degrade over time. Although the performance of the edge version is ideal, it is still unknown how it would scale with a higher parallelization count and with smaller problem instances that could compromise the results.

Figure 7 shows the average time taken by each implementation across all the different runs (with varying number of cores) on all the different parts of the algorithm. The parts include the initialization, the infinite value computation, the relaxation procedure, and the negative cycle detection. As is evident from the figure, the relaxation part is the most time-consuming. This finding highlights the importance of focusing on the relaxation procedure when looking for parallelization opportunities and optimizations. Since this step dominates the total execution time, even small improvements here can significantly impact the overall performance of the algorithm.

Given the critical role of the relaxation part, it's essential to explore various techniques to optimize it. This could include more efficient data structures, better memory management, and advanced parallelization strategies. By scrutinizing this part of the algorithm, we can potentially unlock significant performance gains and make the implementations more efficient.

3.2 CUDA

Similar to Section 3.1, the difference between each run on a given input is negligible (as shown in Figure 8). Therefore, I use the mean time taken by each version on each input as a proxy for its actual runtime.

Figure 9 shows the throughput of both CUDA implementations. The input size is computed as three times the number of edges for the edge version and number of nodes^2 for the node version.

While the edge version generally has a higher throughput, it also exhibits more instability. This instability is likely due to the higher number of checks needed in the edge version, which leads to multiple branches that could desynchronize the cores. In a parallel processing environment, such as with CUDA, maintaining synchronization among cores is crucial for maximizing performance. The desynchroniza-

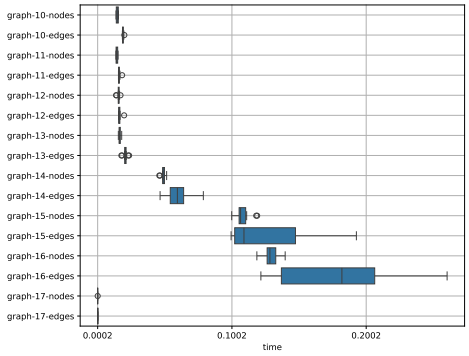

Figure 8: Variation of the execution time for a given combination of input and version on a CUDA device.

tion caused by multiple branching can lead to performance penalties, thus explaining the observed instability.

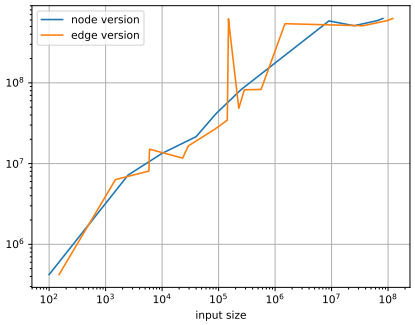

Figure 9: Throughput of the two different versions of the Bellman-Ford algorithm.

Despite this, both versions demonstrate good scalability. As the input size increases, the throughput also improves, which indicates that parallelism helps in both cases. This improvement suggests that both the edge and node versions of the algorithm are well-suited for parallel execution, and they benefit from the increased resources provided by the CUDA architecture.

Finally, Figure 10 shows the speedup of each implementation compared to the sequential version running on the CPU. Speedup, in this case, is defined as the ratio of the time taken by the sequential algorithm to the time taken by the CUDA algorithm, highlighting the performance improvements due to parallel execution.

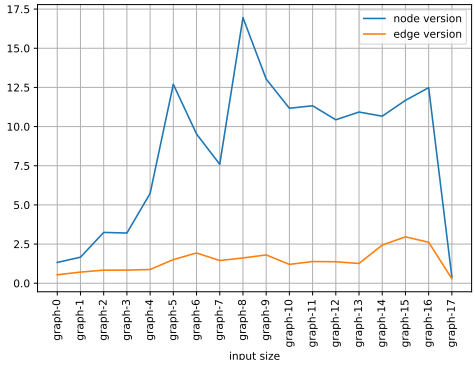

Figure 10: Speedup of the two versions of the CUDA algorithm compared to the sequential version on the CPU.

The results in Figure 10 mirror those observed in Figure 9. The node version consistently demonstrates a higher speedup compared to the edge version. This indicates that the node-based parallel implementation is more effective at leveraging the available computational resources to reduce execution time. The consistently high speedup of the node version suggests that this approach is well-optimized for parallel execution, making it particularly advantageous for large-scale computations.

Both implementations still show consistently better results than the sequential version. This consistent improvement underscores the effectiveness of parallelization for the Bellman-Ford algorithm.

The results depicted in Figure 11 are comparable to those observed in the OMP implementation. In both implementations, the relaxation phase significantly dominates the overall execution time. This phase is inherently more complex and computationally intensive compared to other phases like initialization.

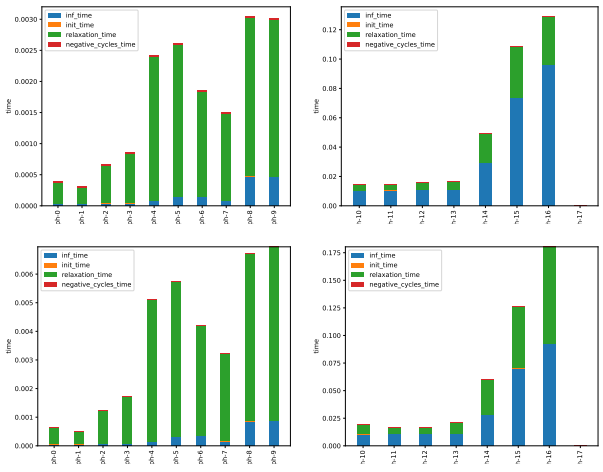

Figure 11: Time taken by each section of the algorithm on the edge (top) and nodes (bottom) versions on CUDA.

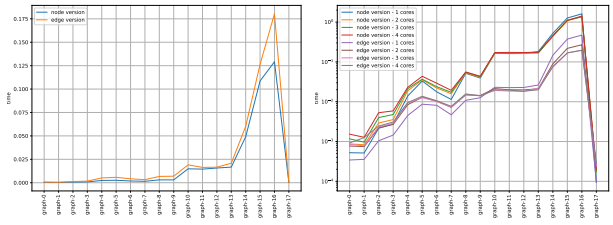

Figure 12: Execution times on CUDA (left) and OMP (right).

4 Conclusions

Both versions of the algorithm scale reasonably well with high input sizes, but performance can suffer for small graphs. This is an important consideration when deciding on the use of parallelization techniques. For smaller graphs, the overhead associated with parallelization may outweigh the benefits, resulting in diminished performance. However, as the input size grows, the benefits of parallelization become more apparent, leading to better scalability and improved execution times.

In each case, the CUDA implementation consistently outperforms the sequential one, even on smaller inputs. This indicates that the Bellman-Ford algorithm is well-suited for GPU implementation.

For the CPU version, maintaining a constant number of threads throughout the algorithm's execution could improve parallelization performance on larger graphs. This approach ensures that the computational resources are fully utilized throughout the execution, leading to better load balancing and reduced idle times. Additionally, it could make parallelization worthwhile even for smaller input sizes. By optimizing the thread management and reducing the overhead associated with dynamic thread allocation, the CPU implementation can achieve better efficiency and performance across different graph sizes.

In summary, both the CUDA and CPU versions of the algorithm have their strengths, but the CUDA implementation shows particular promise for high-performance applications. Further optimization efforts, especially focusing on the factors contributing to performance variability, can enhance the efficiency and scalability of these implementations, making them even more effective for large-scale graph processing.