Architecture and Platforms of AI Project
Analysis of Bellman-Ford on Uber Movement Data

Mohammad Reza Ghasemi Madani
Second Cycle Degree in Artificial Intelligence

Abstract

The proposed study evaluates the performance, speedup, and scalability of the parallel implementations of the Bellman-Ford algorithm on the *DIMACS Challenge Datasets*[1], and discusses challenges specific to parallelizing it using OpenMP and CUDA. The results demonstrate significant reductions in execution time, contributing to the field of parallel graph algorithms and optimizing transportation network analysis.

1 Problem and Motivation

Bellman-Ford's algorithm is popular for finding the shortest paths in a graph from a single source vertex. The sequential implementation of Bellman-Ford's algorithm has a time complexity of $O(|V||E|)$, making it impractical for large-scale graphs. Therefore, as the size of the graph increases, the sequential execution of Bellman-Ford's algorithm becomes a bottleneck.

The motivation behind parallelizing Bellman-Ford's algorithm using OpenMP and CUDA is to improve performance and scalability. By distributing the workload across multiple processing units, parallelization enables faster execution and handling of larger graphs and can significantly reduce the execution time and improve the scalability of the algorithm.

2 Datasets

We performed our experiments with real-world data from the DIMACS Implementation Challenge. These datasets have been meticulously curated to address complex computational problems. These datasets provide a standardized platform for rigorous testing and comparison, facilitating the development of robust and efficient algorithms.

3 Experiments

The Bellman-Ford algorithm implemented is data parallelized. **Data parallelism** involves parallelizing operations on independent data. The parallelization is applied to the relaxation step, where the distances are updated based on the weights of the edges. For instance, the `#pragma omp parallel (for)` is used to distribute the iterations of the loop across multiple threads, allowing them to work on different edges concurrently. In

[1] https://www.diag.uniroma1.it/challenge9/download.shtml

this case, each iteration of the loop represents an independent update of the distance for a specific edge, making it suitable for data parallelism. **Task parallelism**, on the other hand, involves parallelizing different tasks that may have dependencies.

3.1 Implementation

The overall structure of parallel implementation is shown in Figure 1. Each iteration is dependent on the result of the previous iteration. Therefore, we can only start the next iteration when we have relaxed all the nodes' values. Besides, within each iteration of the loop, there is a particular dependency that is each relaxation of an edge (u, v) depends on the result of the last relaxation of an edge directed towards v. Therefore, threads are distributed to update the first $u \in V$ to all other V. We wait for all threads (using `#pragma omp barrier` in OpenMP or `cudaDeviceSynchronize()` in CUDA) to be finished, and then we repeat the process for the next $u \in V$ until we cover all of them. At this point, we have done one single iteration. Now we check if any distance from the source node to others has changed or not. If there is no change we can exit and return results. Otherwise, we try another iteration. In the worst scenario, we must repeat iterations $(|V| - 1)$ times. Where $|V|$ indicates the number of nodes in the graph.

3.1.1 Parallelization Techniques

OpenMP: To parallelize the algorithm, we use a set of directives to distribute loops over CPU cores. Figure 1 shows the general structure of the implemented strategy.

CUDA: Regarding the CUDA implementation, we experiment with a more comprehensive analysis by exploring different aspects of GPUs in parallel optimization. The analysis includes the following aspects:

- `bellman_ford_sequential`: Sequential implementation of Bellman-Ford.
- `bellman_ford_withThread`: Parallelizing using threads only.
- `bellman_ford_withBlock`: Parallelizing using blocks only.
- `bellman_ford`: Leveraging both threads and blocks advantages for parallelization.

3.2 Evaluation Metrics

After implementing different parallelization techniques, we analyze the following metrics:

Speedup is the main measurement metric and represents the acceleration received through parallelization. The speedup, executed with p cores, is calculated as:

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)} \approx \frac{T_{parallel}(1)}{T_{parallel}(p)} \tag{1}$$

However, this is not possible with a CUDA program because we have no power on the number of cores to use nor on the scheduling of threads. The only aspect of the computation that it can control is the size of the number of blocks and threads for each block.

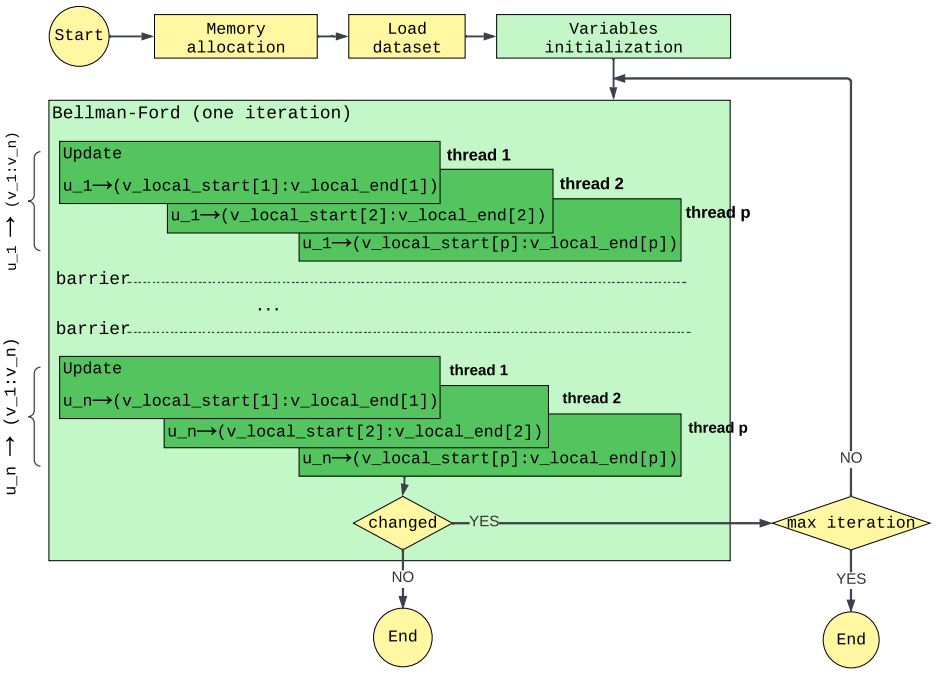

Figure 1: The flowchart of parallel Bellman-Ford

Throughput is a more representative metric for a CUDA application. It consists of the number of "elements" processed in one second. In our case, it is measured as the number of edge relaxations performed per second and it is computed as follows:

$$Throughput = \frac{|E||V|}{T_{CUDA}} \tag{2}$$

Scalability checks if it efficiently handles larger datasets and more threads.

Efficiency indicates how parallelization is efficiently utilizing the available threads.

$$E(p) = \frac{S(p)}{p} = \frac{T_{parallel}(1)}{p \times T_{parallel}(p)} \tag{3}$$

Amdahl's Law Ideally, we would like to achieve linear speedup in our program, regardless of the number of processors used. However, the fraction α of the program increases times for **synchronization** and **communication**, in addition to the overhead of allocating the necessary memory. For this reason, the speedup is always less than p.

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)} = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{4}$$

3.3 Instruction

To run the program, use `sbatch project.sbatch` command. This executes three files: i) `bellman_ford_omp.c`, ii) `bellman_ford_cuda.cu`, and iii) `compare_omp_cuda.c`. To specify the `N_THREADS` for OpenMP or `<<<grids, blocks>>>` for CUDA, modify them in `bellman_ford_omp.c` and `bellman_ford_cuda.cu`. Since the dataset is very large

(264,346 nodes, 733,846 edges) we have the option to select a smaller subset by setting `VERTICES` at the beginning of each program. In this case, we only extract nodes and edges with node IDs less than `VERTICES` value as well as their associated edges. After successful execution, `omp_output.txt` and `cuda_output.txt` are generated and include the results for each program. Also, the `report.out` gives a summary of the input specifications (number of nodes and edges), OpenMP and CUDA configuration (number of threads, blocks, and threads) and runtimes, and verification of the results by comparing the OpenMP and CUDA outputs through `compare_omp_cuda.c` (showing number of mismatch results).[2]

4 Results

We experimented with `VERTICES` $\in \{5000, 10000, 15000, 20000, 25000\}$. It is important to consider the fact that sub-sampling the dataset may affect the density of the network or create disjoint graphs which may affect execution time. Table 1 indicates the `VERTICES`, `EDGES` (based on selected `VERTICES`), and `N_THREADS` selected for each experiment. To obtain the maximum number of processors for OpenMP, `nproc` command returns the number of available processors in the system. In our case, there are 2 processors, so we can not exceed `N_THREADS=2`. To measure the sequential execution time, we set `N_THREADS=1`. Using CUDA, to specify `<<<grids, blocks>>>` (when using `bellman_ford_withBlock` and `bellman_ford`), we pass `BLKDIM` as input parameter and compute them based on: `grids` $= \frac{\texttt{(N+BLKDIM-1)}}{\texttt{BLKDIM}}$ and `blocks` $=$ `BLKDIM`. Besides, we set the `START` node to 2978 since this node has the most number of outgoing edges (which is 6) for all the specified `VERTICES` and makes the problem more challenging.

4.1 Analysis

Table 1 presents the results of the OpenMP implementation for different configurations. As the number of vertices increases, the execution time also increases, which is expected since the algorithm's complexity is O(|V||E|). When comparing the execution times between a single thread and 2 threads, a modest `Speedup` is observed, around 1.9. However, the `Efficiency` hovers around 0.95, indicating that the parallelization efficiency is relatively high, considering the overhead introduced by managing multiple threads. Overall, the results suggest that the OpenMP parallelization provides reasonable speedup and efficiency for the Bellman-Ford algorithm across different problem sizes, making it a promising approach for larger graphs.

Table 1: OpenMP results

VERTICES	EDGES	THREADS	Time (sec)	Speedup	Efficiency	Alpha
5000	12202	1 2	7.089554 3.664526	1.93	0.97	0.03
10000	24116	1 2	30.78094 15.836158	1.94	0.97	0.03
15000	37510	1 2	69.397323 36.447641	1.90	0.95	0.05
20000	50796	1 2	123.525761 65.429852	1.89	0.94	0.06
25000	65210	1 2	194.391969 104.491675	1.86	0.93	0.07

[2]https://github.com/qasemii/Bellman-Ford.git

Table 2 displays the results of the CUDA implementation for different settings. The sequential implementation serves as a baseline. For each `Type`, the number of `grids` and `blocks` is specified. The Thread/Block approach, combining both thread and block parallelism, exhibits the best performance across all scenarios by significantly reducing the runtime. This underlines the importance of leveraging both thread and block parallelism for achieving superior performance in CUDA implementations of the Bellman-Ford algorithm. The Thread Only approach, and Block Only approach, also demonstrate significant speedup compared to the sequential implementation.

Table 2: CUDA results

VERTICES	EDGES	Type	grids	blocks	Time (sec)	Throughput (M/sec)
5000	12202	Sequential	1	1	101.394509	0.60
		Thread Only	1	128	4.659423	13.09
		Block Only	5000	1	0.660597	92.36
		Thread/Block	40	128	0.248434	245.58
10000	24116	Sequential	1	1	445.343621	0.54
		Thread Only	1	128	19.153588	12.59
		Block Only	10000	1	2.721449	88.61
		Thread/Block	79	128	0.452966	532.40
15000	37510	Sequential	1	1	996.619589	0.56
		Thread Only	1	128	42.849908	13.13
		Block Only	15000	1	5.442502	103.38
		Thread/Block	118	128	0.707938	794.77
20000	50796	Sequential	1	1	1773.362727	0.57
		Thread Only	1	128	77.298262	13.14
		Block Only	20000	1	9.994378	101.65
		Thread/Block	157	128	0.987004	1029.30
25000	65210	Sequential	1	1	2771.418733	0.59
		Thread Only	1	128	122.797174	13.28
		Block Only	25000	1	14.082969	115.76
		Thread/Block	196	128	1.40301	1161.97

Finally, looking at Figure 2 we can see the by increasing the size of the input, for all types `Throughput` remain slightly the same except Thread/Block. In this case, `Throughput` increases by increasing the size of the input.

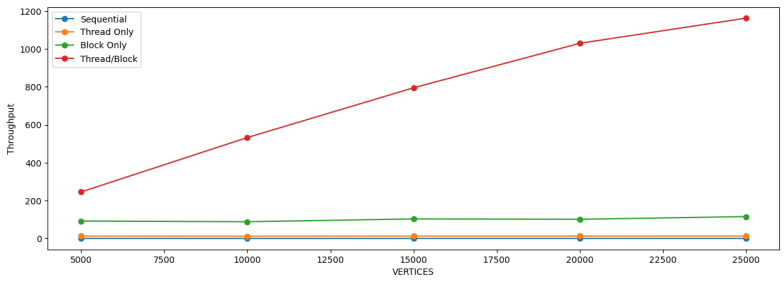

Figure 2: Throughput over number of VERTICES

5 Conclusion

In conclusion, the problem of long computation time and the need for scalability in Bellman-Ford's algorithm motivate the exploration and development of parallel implementations using OpenMP and CUDA. By harnessing the power of parallel processing, we can unlock substantial performance improvements and enable the algorithm to handle increasingly complex graph-based problems.