Parallel implementation of Bellman Ford algorithm
Architectures and Platform for AI exam project

Alessio Conti
Master's Degree in Artificial Intelligence, University of Bologna
alessio.conti3@studio.unibo.it

Abstract

The Bellman-Ford algorithm is a well-known solution to the problem of finding the shortest paths in a graph from a single source to all other vertices (SSSP problem). This project presents three different parallel implementations of the Bellman-Ford algorithm, using both OpenMP directives for CPU parallelisation and CUDA C programming techniques for GPU architectures. The aim is to improve Bellman-Ford performance and eventually work efficiency of the algorithm. The experimental results show that all three proposed implementations provide large improvements over the sequential version of the algorithm.

The best one ($omp\_frontier$) achieves an average speedup of more than 200x times higher than the omp_base sequential implementation. When run on the maximum available core instead, it performs two times faster than its own sequential counterpart.

1 Introduction

The addressed task wants to resolve the single source shortest paths (SSSP) problem. The aim is to find the shortest path from a single given vertex to all the others in the graph. The fame and importance of this well-known problem is given by the various domains it can be applied at, such as: road networks, routing protocols, social networks, data mining and artificial intelligence. Two algorithms are mainly adopted in this scenario: Dijkstra and Bellman Ford. For this project, the focus is testing different parallel implementations of Bellman Ford's algorithm. Bellman Ford's algorithm, in its sequential form, is based on an iterative process over all edge connections. Each iteration updates the vertices continuously if a better route is found until the final distances converge (Cormen et al., 2001). The algorithm's strength lies in its ability to process graphs containing negative weight edges and to identify graphs with negative cycles. This represents a significant advantage for the algorithm, enabling its application in scenarios where negative weights are inherent to the domain of the modelled data. Such scenarios include power allocation in wireless sensor networks (Zhang et al., 2014), systems biology, and regenerative braking energy for railway vehicles (Lu et al., 2014).
This project presents three distinct parallel implementations of the algorithm. Two of these are CPU-based and have been developed using OpenMP, while the third is a GPU-based implementation and has been developed in CUDA.
All three approaches preserve the semantics of the algorithm.

2 Background

This section summarizes concepts on *OpenMP*, *CUDA*, *Bellman-Ford* algorithm and the notion of *Frontier*.

2.1 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming. The principal functionalities include directives for parallel regions, loop parallelism, task parallelism, and synchronization constructs such as barriers and locks. The OpenMP standard enables parallelism using pragmas: compiler directives that specify parallel regions and how they should be executed across threads. Pragmas facilitate application scaling in multiprocessor systems, ensuring correct utilisation of the available cores. Furthermore, OpenMP offers runtime library routines for dynamic thread management, data sharing, and performance monitoring (OpenMP, 2024).

2.2 CUDA

CUDA (Compute Unified Device Architecture) is a C library extension developed by NVIDIA with the objective of providing a programming interface

to GPU devices. The CPU (host) is responsible for initiating the main program and executing serial code, while delegating the execution of compute-intensive tasks to the GPU (device). The functions designated to execute in parallel by multiple GPU threads are referred to as kernels.

2.3 Bellman-Ford algorithm

Given a graph $G(V, E)$ (directed or undirected) the algorithm's aim is to find the shortest path possible between a source vertex s and every other vertex of V considering a weight function $w : E \rightarrow R$ associated to each edge.

```
for all vertices u in V(G) do:
   d(u) = inf
d(s) = 0
for all edges (u, v) in E(G) do
   relax(u, v, w)
```

At each iteration of the algorithm each edge (u, v) with weight w, goes through a Relax operation. This operation verifies if is possible to improve the distance $d(v)$ from u to v already found.

```
relax(u, v, w):
   if d(u) + w < d(v) then:
      d(v) = d(u) + w
```

The algorithm updates the distance value of each vertex continuously until the final distances converge or until the maximum number of iterations V is reached, meaning the graph presents a negative weight cycle. This is because in a graph with V vertices can not exist a shorter path that contains at most V edges.

2.4 Frontier

Furthermore, the concept of frontier, employed in one of the algorithm's implementations, is worthy of mention. Frontier is typically understood to be the set of active nodes that an algorithm is currently handling. Initially, the frontier consists solely of the starting node. At each iteration, the algorithm expands the frontier by adding vertices that must be explored further. This approach enables the algorithm to efficiently handle only the nodes of interest, thereby eliminating the need to consider any unnecessary ones.

3 System description

The project compares three different algorithms: omp_base, $omp\_frontier$, and $cuda\_base$.
The former two algorithms are developed using OpenMP, while the latter is developed on CUDA C.

3.1 OpenMP Base

Omp_base algorithm is the simplest of the three and follows the structure of the serial Bellman Ford implementation. The algorithm employs parallelisation at two distinct stages: firstly, for the initialisation of the data structure handling the tentative distance between s and the other vertices, and secondly, for the execution of the Relax operation. This is possible because, at each iteration, the algorithm verifies whether it is possible to improve the tentative distance between any two connected nodes u and v, given the corresponding weight w. This operation can be performed concurrently, as there is no specific ordering required and no dependencies between edges.
The only operation that must be performed in a critical section are the assignment of an improved distance, ensuring that only one thread at a time writes the array.
To detect the presence of negative cycles the algorithm performs a series of $V - 1$ iterations. Then another one (the Vth iteration) is performed. During the latter, it checks for any changes occurred in path's distances to exclude the presence of negative weight cycles.

3.2 OpenMP Frontier

The second version of the algorithm developed in OpenMP ($omp\_frontier$) uses the concept of frontiers. The complexity of the algorithm is determined by the number of Relax operations performed using an atomic instruction to handle race conditions. This instruction is more time-consuming than ordinary memory access. Therefore, reducing the number of Relax operations to only those that are absolutely necessary at each iteration would improve the performance of the algorithm. $Omp\_frontier$ tries to achieve this by using two frontier data structures, F1 and F2 (Busato and Bombieri, 2016).
In this scenario, active vertices are those whose tentative distance has been modified, only these need to be considered for the relax procedure in the next iteration. If the tentative distance $d(v)$ does not change during iteration i, there is no need to relax any edge outgoing from v in iteration $i + 1$. Thus, v is not inserted into the frontier.
At each iteration i, the algorithm concurrently extracts all the nodes from F1, for each of those it concurrently performs a Relax operation on all the outgoing edges. If a tentative distance changes, the

newly reached node is inserted in F2 (atomically). At the end of each iteration a swap between F1 and F2 is performed.
In order to increase efficiency and eliminate redundancies within the frontier nodes, all frontiers insertion avoids duplicates. The algorithm runs until the frontier is empty or it reaches V iterations, which signals the presence of negative weight cycles.
The parallel processing performed on active nodes ensures the semantic coherence of the algorithm. This is due to two factors: each node's processing is independent of the others; and including non-active nodes in the processing phase does not affect the result. In the worst case, edges can be added $O(|E|)$ times into the frontier.
The frontier data structure is built as a circular list that can be continuously emptied and refilled up to $|E|$. It is developed considering its use within parallel threads, therefore any operation that affects the elements contained in the frontier is conducted within a critical section.

3.3 CUDA Base

The final implementation of the algorithm ($cuda\_base$) is developed in CUDA C and resembles the same structure of omp_base. The algorithm is conducted by the host machine. After allocating and initialising the structures, both in the host machine and in the device, the host machine runs $V - 1$ iteration of the algorithm. At each iteration i, all the E edges must be relaxed. This is achieved by invoking a kernel function that handles the relax operation on the device. The workload ($O(|E|)$) is distributed among different blocks (number dynamically changes depending on $|E|$) of size 256. The kernel utilises an atomic instruction in order to compute and store the minimum between two values. Before moving on to the next iteration, it is necessary to ensure that the device has completed all preceding tasks. Once all the initial $V-1$ iterations have been completed, the host runs another iteration to exclude the presence of negative cycles, as in omp_base.

4 Data

The three algorithm's versions are tested on four different graphs of different sizes, in order to test scalability. All four graphs are generated to be directed graphs and do not contain negative cycles, even if they may present edges with negative weights.
In the following table are synthesized the graph's dimensions in number of vertices and edges.

graph	Vertices	Edges
graph_XS	250	1.5k
graph_S	500	10k
graph_M	1.5k	200k
graph_L	10k	500k

Table 1: Graph's dimensions.

The graphs are stored as plain text file as adjacency matrix. The first two rows of each file contains respectively the number of vertices and edges present in the graph.

5 Experimental setup and results

Each algorithm has been tested with each of the four graphs. Each experiment has been repeated five times, to extract an average execution time.

5.1 OpenMP results

For what concerns omp_base and $omp\_frontier$ the experiments were performed with different numbers of threads ranging from 1 to four time the physical number of threads in the machine. Scaling by the doubling from one experiment to the other. In the scenario of Slurm Cluster installed at the DISI - University of Bologna, which offers 4 CPU cores, the experiments were run with $[1, 2, 4, 8, 16]$ threads. Cuda tests are performed on a Turing GPU.

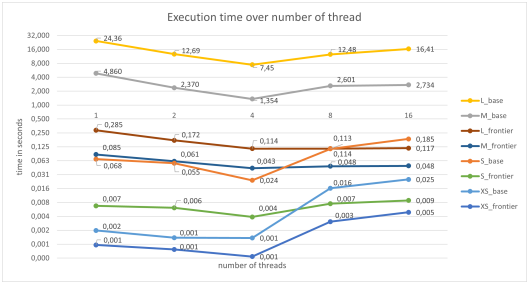

Figure 1: Timing for omp_base and $omp\_frontier$ over multiple graphs size and different thread usage.

The first experiment wanted to test the algorithm speed between the two OpenMP implementations omp_base and $omp\_frontier$. The results are displayed in Table 2. It is evident that $omp\_frontier$ outperforms omp_base in a multitude of scenarios. In the initial two tests with relatively small input, the algorithm demonstrated a notable speed

threads	graph_XS		graph_S		graph_M		graph_L	
	base	***frontier***	***base***	***frontier***	***base***	***frontier***	***base***	***frontier***
1	0.0019	0.0009	0.0678	0.0065	4.8601	0.0853	24.3596	0.2853
2	0.0013	0.0007	0.0552	0.0060	2.3697	0.0610	12.6895	0.1723
4	0.0013	0.0005	0.0236	0.0037	1.3544	0.0431	7.4458	0.1141
8	0.0158	0.0029	0.1131	0.0073	2.6005	0.0475	12.4823	0.1141
16	0.0246	0.0048	0.1846	0.0086	2.7342	0.0484	16.4062	0.1171

Table 2: Time comparison of OpenMP algorithm omp_base and $omp\_frontier$ (in seconds).

advantage. The difference reached a factor of one order of magnitude. Conversely, in tests involving larger graphs, the performance gap between the two algorithms widened, reaching a difference of two orders of magnitude.

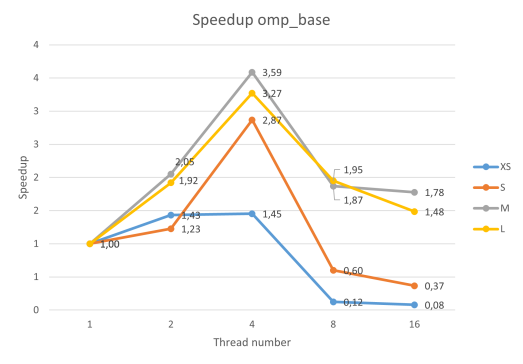

Figure 2: Omp_base speedup comparison among different thread usage.

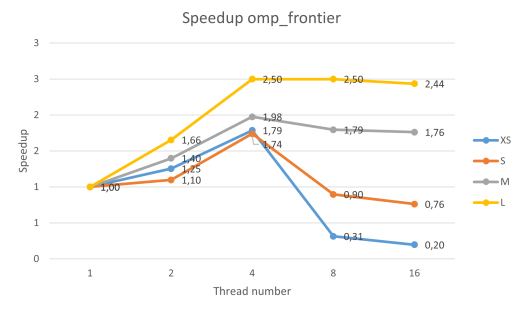

Figure 3: $Omp\_frontier$ speedup comparison among different thread usage.

The experiments are conducted by increasing the number of threads beyond the actual physical available threads of the machine (four). For each of the experiments, the fastest execution time was observed when four CPU cores were utilised, both for omp_base and $omp\_frontier$. With fewer threads, the algorithm performs poorly given the high amount of work each core has to handle. Above four instead, the operating system must switch between those. In the event that there are more threads than available cores, the operating system is compelled to frequently switch between threads. This results in frequent overhead from context switching, having a significant impact on performance. In some cases ($graph\_XS$ and $graph\_S$), the algorithm performs even worse than in sequential execution. This can be seen in Figure 2, Figure 3.

In order to understand the performance increase of one algorithm version among the change of threads number, speedup is compute as follow.

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

where p is the number of threads used.

As illustrated in Table 3, the speedup with respect to sequential execution is considerable, reaching peaks of almost 2 points (and occasionally exceeding this value) when executing both algorithms with two and four cores.
For the smaller graphs, the performance deteriorates rapidly as the number of threads is increased. In contrast, for larger graphs such as $graph\_M$ and $graph\_L$, the speedup remains advantageous, despite the performance not being optimal. Nevertheless, it is still superior to the corresponding sequential implementation.

The efficacy of OpenMP algorithms was also evaluated in terms of strong scaling efficiency. Efficiency is calculated as:

$$\frac{Speedup(p)}{p}$$

where p is the number of threads used.
The results are presented in Table 4. It was observed that the efficiency of both omp_base and $omp\_parallel$ exhibited a similar decline, falling below 0.5, after the number of threads was increased over the maximum number of physical cores available.

threads	graph_XS		graph_S		graph_M		graph_L	
	base	***frontier***	***base***	***frontier***	***base***	***frontier***	***base***	***frontier***
2	1.434	1.253	1.226	1.097	2.050	1.398	1.919	1.655
4	1.453	1.785	2.867	1.739	3.588	1.975	3.271	2.499
8	0.122	0.314	0.599	0.897	1.868	1.794	1.951	2.500
16	0.078	0.196	0.367	0.762	1.777	1.761	1.484	2.435

Table 3: Speedup between OpenMP algorithms.

threads	graph_XS		graph_S		graph_M		graph_L	
	base	***frontier***	***base***	***frontier***	***base***	***frontier***	***base***	***frontier***
2	0.717	0.626	0.613	0.548	1.025	0.699	0.959	0.827
4	0.363	0.446	0.716	0.434	0.897	0.493	0.817	0.624
8	0.015	0.039	0.074	0.112	0.233	0.224	0.243	0.312
16	0.004	0.012	0.022	0.047	0.111	0.110	0.092	0.152

Table 4: Efficiency between OpenMP algorithms.

5.2 CUDA results

graph	*cuda_base*
graph_XS	0.0015
graph_S	0.0095
graph_M	0.0479
graph_L	0.1541

Table 5: Timing of $cuda\_base$ algorithm on GPU.

The objective of the second experiment was to assess the efficacy of implementing $cuda\_base$ on the GPU. The tests were repeated five times, and the resulting average execution time was calculated. Despite the algorithmic structure of $cuda\_base$ bearing resemblance to that of omp_base, the outcomes are found to be in close proximity to those achieved by $omp\_frontier$.

graph	cuda vs omp_base	cuda vs omp_frontier
graph_XS	1.252	1.643
graph_S	7.116	1.444
graph_M	101.3	0.562
graph_L	157.9	0.540

Table 6: Speedup comparison of $cuda\_base$ against omp_base and $omp\_frontier$.

$Cuda\_base$ speedup performance are compared against both omp_base and $omp\_frontier$ sequential baselines.
Table 6 reveals a significant improvement of over one hundred points for $graph\_M$ and $graph\_L$ when looking at speedup performance compared to omp_base. This discrepancy is reduced when the comparison is made with $omp\_frontier$, where the timing discrepancy is also lower, as stated in Table 5. For large input graphs ($graph\_M$ and $graph\_L$) $cuda\_base$ has comparable performance to $omp\_frontier$, even considering the latter at its maximum potential leveraging 4 cores.

6 Discussion

As previously stated, all three approaches resulted in a notable enhancement of the standard Bellman-Ford sequential algorithm implementation. The $omp\_frontier$ algorithm is the fastest of the proposed algorithms. This outcome is a consequence of both parallelisation and the optimisation implemented to limit the inefficiency typical of the Bellman-Ford algorithm. This is particularly evident in Table 7, where it can be observed that each algorithm exhibits a notable improvement with respect to the sequential execution of omp_base. The algorithm achieving higher speedups for all input dimensions is $omp\_frontier$. The speedup continues growing when increasing the input graph size; this is a sign of great potentiality, especially for $omp\_frontier$ and $cuda\_base$, meaning they can solve larger problems with minimal time overhead, due to their efficient resource management.

It is noteworthy that even a modest investment in parallelisation can yield significant enhancements to the algorithm's overall performance. One illus-

graph	omp_base	omp_frontier	cuda_base
graph_XS	1,453	3,678	1,252
graph_S	2,867	17,88	7,116
graph_M	3,588	112,5	101,3
graph_L	3,271	213,4	157,9

Table 7: Speedup comparison of the three algorithms executed with optimal thread number, against omp_base sequential as the baseline.

trative example is omp_base, which has not undergone any optimisation. This algorithm exhibits a speedup improvement of over 3.2x times for the two larger input graphs, Table 3 and Table 6. Undergoing optimization techniques instead helps the algorithm to perform far better, reaching speedups improvement of more than 200x times for $omp\_frontier$ tested on $graph\_L$ Table 7. Even if $cuda\_base$ implementation is simple and absent of optimization techniques, the performance obtained through the utilisation of GPU power is comparable to that of $omp\_frontier$. This represents a significant opportunity for the future implementation of a CUDA version of $omp\_frontier$. The combination of algorithmic optimisation and GPU power could result in significant enhancements.

7 Conclusion

The project presented different implementations of the Bellman-Ford algorithm, giving focus mainly to parallelization techniques but also keeping attention to algorithmic optimizations. All this to improve the performance and at the same time, to optimize the work inefficiency typical of the Bellman-Ford algorithm.

Experimental results have been conducted on graphs of different sizes and the comparisons are made with respect to the algorithm's sequential counterpart. Finally, the work presented an analysis of the proposed strategies to understand the impact of different parallelization techniques. As future improvements of the project would be interesting to develop a CUDA version of $omp\_frontier$, that employs frontier to reduce the number of Relax operations. It would be also beneficial to take advantage of more sophisticated and lightweight data structures to fully utilise the fiscal capabilities of the underlying architecture. Then it would be interesting to test those algorithms with standard benchmarks and see where they rank.

References

Federico Busato and Nicola Bombieri. 2016. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*, 2nd edition. The MIT Press.

Shaofeng Lu, Paul Weston, Stuart Hillmansen, Hoay Beng Gooi, and Clive Roberts. 2014. Increasing the regenerative braking energy for railway vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 15(6):2506–2515.

OpenMP. 2024. [link].

Xinming Zhang, Fan Yan, Lei Tao, and Dan Keun Sung. 2014. Optimal candidate set for opportunistic routing in asynchronous wireless sensor networks. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8.