A Parallel Implementation of the Bellman-Ford Algorithm Using OpenMP and CUDA

Angelo Galavotti
angelo.galavotti@studio.unibo.it

September 18, 2024

1 Introduction

This document shows a possible implementation of the Bellman-Ford algorithm, with a thorough analysis on the performance using different threads configurations and graphs of varying sizes. To achieve this, two programs were developed using the following interfaces:

- OpenMP [1], a specification for parallel programming which allows to easily develop multi-threaded applications.
- CUDA [2], which provides a set of instructions that allow the execution of parallel tasks on NVIDIA GPUs.

Since each of them exploits the parallelization capabilities of the CPU and GPU respectively, the results highlight the advantages and drawbacks of using these two different technologies.

1.1 Bellman-Ford algorithm

The Bellman-Ford algorithm [3] was proposed in 1956 by researchers Richard Bellman and Lester Ford Jr., and allows to find the shortest path from a single vertex of a weighted graph. Its key feature is the ability to handle negative weighted edges.

The general algorithm functions as follows:

```
def BellmanFord(G, S, W):
    D <- a distance array with distances for each node
    D[S] <- 0
    R <- V - {S}
    C <- cardinality(V)

    # Initialize distances array
    for vertex k in R:
        D[k] <- infinity

    # Relax edges repeatedly
    for vertex i from 1 to (C - 1):
         for edge (e1, e2) in E:
             # Relax the edge if necessary
             if (D[e1] != infinity &&  D[e2] > D[e1] + W[e1, e2]:
                    D[e2] <- D[e1] + W[e1, e2];

    # Check for negative weight cycles
    for edge (e1, e2) in E:
        if D[e2] > D[e1] + W[e1, e2]:
            print("Graph contains negative weight cycle")
```

The time complexity of the algorithm is $O(|V| \cdot |E|)$, where V is the number of nodes and E is the number of edges.

Like other SSSP algorithms (e.g. Dijkstra), it involves a relaxation step, in which a non-optimal solution is computed, and improved upon until the optimal solution is found, i.e. the are no other ways to refine the solution. In the Bellman-Ford algorithm, this relaxation step is performed $|V| - 1$ times. This is because the shortest path between any two vertices can have at most $|V| - 1$ edges.

2 Implementation

The instructions of the algorithm that were reworked in order to allow for parallelization were essentially each one of its looping parts. Namely:

- The initialization of the distance array.
- The relaxation step.
- The negative weight detection step.

The broad idea for the structure of the parallelised code was the same for CUDA and OpenMP. The first loop is embarrassingly parallel, as it modifies parts of the distance array that are completely separate from each thread. The same cannot be said for the relaxation step, which requires some kind of barrier when updating the distance array value. Finally, the negative weight detection step requires the value assignment of a ''negative check" flag, since none of the APIs used allow to exit from a function using the `return` command.

For most of the development, the graph was implemented as an adjacency list, as it is one of the most popular ways of implementing these data structures and allows more versatility when extracting the actual path from the source to the destination. Ultimately, it was decided to refactor the code and adopt a "list of edges" graph representation in order to ease the process of memory management, especially when dealing with the CUDA implementation.[1].

2.1 OpenMP Implementation

The choice of scheduling in OpenMP can greatly impact the efficiency of parallelized tasks. In this project, a static scheduling approach was used over dynamic scheduling, as it showed much better performance the during experiments. In addition, considering that static scheduling divides the workload among threads upfront, it reduces the overhead when dealing with thread initialization.

As previously mentioned, for the relaxation step some kind of way to enforce atomic operations is required. As such, the `critical` operator provided by OpenMP is used. The implementation is shown in the following code snippet.

```
...
if (dist[u] != INT_MAX && (dist[u] + weight < dist[v])) {
#pragma omp critical
    {
        // correctness test
        if (dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
        }
    }
}
...
```

Another approach could see rewriting the critical section region to include the outer `if` operation. However, this would lead to a higher computation time, as all threads will end up performing a

[1]The old implementation can still be found in the public repository of the project in the `old` folder at `https://github.com/AngeloGalav/parallel-bellman-ford`

portion of the code sequentially. Instead, the code presented significantly reduces overall runtime, as fewer threads will block the others to execute any sequential instructions.

Nevertheless, such approach requires a "correctness test", which may seem redundant at first glance. Yet, it is rather vital as it enforces the operation to go through only if the previous conditions are still met, in case another thread has modified the distance vector before the initialization of the critical section.

Finally, for checking negative edges, a shared variable which functions as a "negative check" is used, and its assignment is performed using a single `atomic write` operation. This method ensures that only one thread can write to it at any given time, guaranteeing thread safety without the need for more complex synchronization mechanisms.

2.2 CUDA Implementation

As previously mentioned, the CUDA and OpenMP implementations share a considerable amount of similarities. The "`parrallel for`" instructions were replaced by a respective kernel that allowed for each thread to execute the looping instructions.

In CUDA, blocks and threads are fundamental concepts that help divide the workload. A block is a collection of threads that run concurrently, while threads within the block are identified by a thread ID. In this implementation, the number of blocks were computed based on the graph size, with the formula `(N + BLOCK_SIZE - 1) / BLOCK_SIZE`, where `BLOCK_SIZE` is the size of block, and `N` is either the number of edges E or the number of vertices V, depending on the kernel. The block size for each kernel was set to 512, as this value proved to be the most effective after some experimentation.

To maintain a semantic equivalence with the "correctness test" found in the OpenMP program, CUDA's `atomicMin` operator was used. This operator compares two arguments and assigns the minimum value to the first operand, all while performing these actions atomically. Initially, a different approach was employed, which involved the use of the `_syncthreads()` operator as a barrier, however the `atomicMin` operator allows for a cleaner and simpler implementation.

To allow a comparison between the parallel and serial versions of the Bellman-Ford algorithm on the GPU, sequential variations of the kernels were implemented. These can be executed by setting a specific flag when running the program, and are initialized using a single block of size 1.

When launching the program, the total execution time is measured without considering the memory transfers between CPU and the GPU data, as it is completely unrelated to the performance of the algorithm itself. Additionally, two versions of the timer function `cuda_gettime()` were implemented to account for different operating systems[2]. The Unix-compatible version is based on the `hpc.h`[3] library by Moreno Marzolla [4], while the Windows-compatible one was developed using the `windows.h` and WINAPI functions.

2.3 Experimental setup

The programs were evaluated using graphs of different dimensions (including a graph with negative edges), generated through the Erdős-Rényi method: given N vertices, each pair of vertices is connected with a probability p. The specifications of each graph used during the experiments are detailed in Table 2.3. To thoroughly verify the correctness of the proposed solution, the algorithm's results were compared against those produced by other existing implementations.

The tests were run with the assumption that each sample graph is bi-directed. However, each program allows for testing of directed graphs by specifying the respective flag when launching them.

[2]This allows to test the programs in a cross-platform way before submitting the job on the Slurm cluster. Both the OpenMP and CUDA implementations run seamlessly on Windows and Linux.

[3]The library is part of the material from the High Performance Computing course by Moreno Marzolla at University of Bologna.

Graph Name	Number of Nodes	Number of Edges
`graph_0.txt`	10	25
`graph_1.txt`	100	250
`graph_2.txt`	1000	2500
`graph_3.txt`	5000	10000
`graph_4.txt`	10000	25000
`graph_5.txt`	50000	75000

Table 1: The graphs that were used to test the models.

3 Results

What follows in this section is a detailed analysis of the results obtained from each implementation, accompanied by visual representations. The performance and scalability of both CUDA and OpenMP approaches are evaluated by comparing execution times across various thread configurations.

Additionally, there will be a comparison between the OpenMP and the CUDA implementations, in order to highlight the advantages and disadvantages of using the CPU and GPU for parallelization.

3.1 OpenMP Results

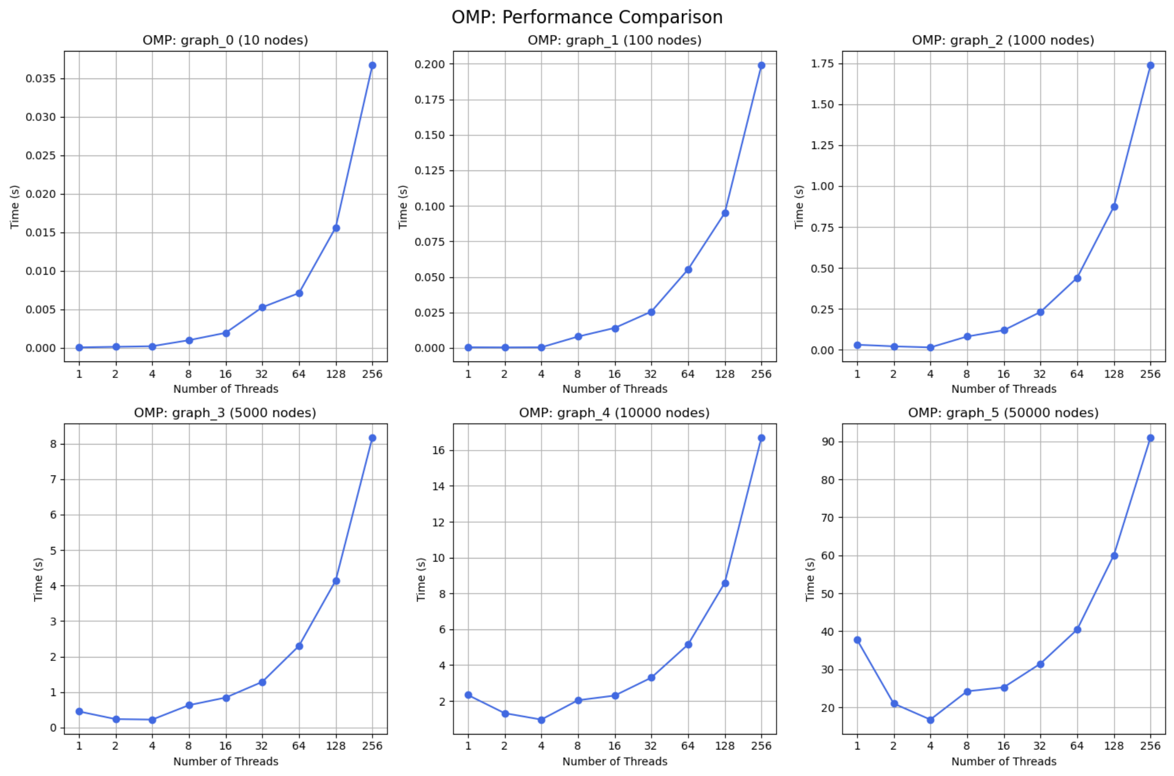

Figure 1: Performance comparison graphs using the OpenMP implementation with different threads.

The results of the OpenMP implementation across various thread configurations are illustrated in Figure 1. In the first set of graphs, it is apparent that using a lower number of threads results in faster execution times. This is due to the fact that as the number of threads increases, more time is spent on overhead instructions such as thread initialization and management. Particularly, for these smaller graphs, the overhead time is larger than the actual execution time of the algorithm, and as a result, the benefits of parallelization are basically negated.

However, starting from `graph_2` (1000 nodes), the execution times decrease significantly compared to the single-threaded version, and there is a notable minima when using around 4 threads. Beyond this point, increasing the thread count leads to diminishing returns. This trend is highlighted in the results on `graph_5` (50,000 nodes), where there is a severe performance degradation when more than 32 threads are used.

The main takeaway from these results is that for smaller graphs, the use of multiple threads introduces too much overhead, while for larger graphs, parallelism can be beneficial only up to a certain point. Beyond that, the added overhead begins to outweigh the performance gains, causing execution time to increase with more threads.

3.2 CUDA Results

The results of the CUDA program across different graph sizes and thread configurations are shown in Figure 2. At first glance, one might observe similar trends between the parallel and serial configurations. However, the difference in performance becomes quite pronounced as graph size increases, with the parallel execution outperforming the single-threaded configuration by several orders of magnitude. This is highlighted in the third graph, where the parallel and serial configurations are directly compared.

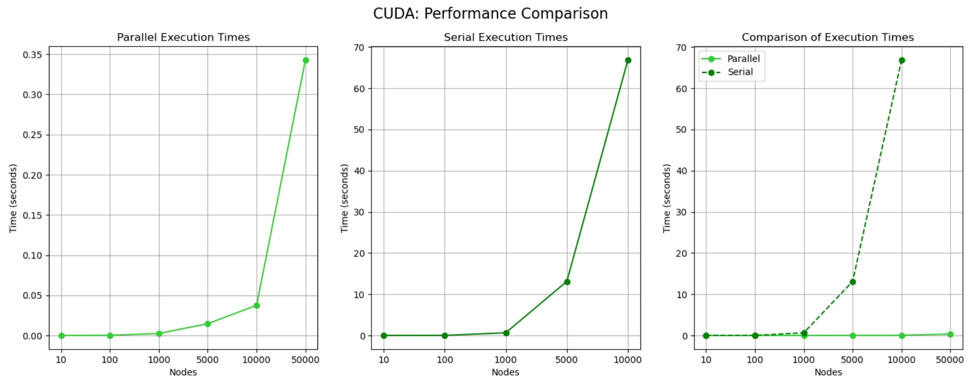

Figure 2: Performance comparison graphs using the CUDA implementation. There is a clear difference between the parallel and one-threaded configurations.

As the graph size grows, the serial execution time increases dramatically, even timing out on the largest graph. In contrast, the parallel version remains highly efficient, handling even `graph_5` (50,000 nodes) in a fraction of the time compared to the sequential configuration.

Therefore, the results show that the GPU's architecture is, unsurprisingly, inherently better suited for parallel processing. Trying to run a sequential algorithm on a GPU severely dampens its performance. As such, it is far superior for handling tasks that can be broken down into smaller, independent subtasks.

3.3 OpenMP vs. CUDA

When comparing the results between the two implementations, it becomes evident that CUDA consistently outperforms OpenMP in terms of execution time, in particular as the graph sizes increase. The GPU's parallel architecture allows for significantly faster processing, making CUDA the better option for handling this type of problem. In addition, the time spent on overhead instructions is minimal in CUDA, as thread management is hardware-accelerated. For this reason, it handles thread initialization, scheduling and synchronization much more effectively than OpenMP.

However, this advantage does not extend to single-threaded tasks. In this situation, the CPU consistently exceeds the GPU's capabilities, with the GPU even timing out on `graph_5` (50,000 nodes).

As mentioned in the previous section, this is inline with the idea that GPU's main function is to handle simple and repetitive tasks in parallel. Its cores are generally slower and have less memory than CPU cores [6], and are not suited for larger sequential operations.

3.4 Conclusion

In this project, the Bellman-Ford algorithm was implemented using two parallelization frameworks: OpenMP for multi-threaded CPU execution and CUDA for GPU-based execution. The analysis of experimental results showed that OpenMP exhibits performance benefits for larger graphs up to a certain threshold, after which the overhead of thread synchronization and initialization outweighs the benefits of parallelism.

On the other hand, CUDA demonstrated superior performance, especially for larger graphs. However, the single-thread performance of the CPU is consistently better than the GPU, and the CUDA implementation faced increasing performance degradations when dealing with larger graphs.

In summary, the Bellman-Ford algorithm benefits the most from parallelization when using larger graphs, and when executing the algorithm on the GPU. Nevertheless, the appropriate number of threads should be carefully chosen to avoid performance degradation.

References

OpenMP Architecture Review Board (2024), *OpenMP Documentation*,
`https://www.openmp.org/`

NVIDIA (2024), *CUDA Documentation*,
`https://docs.nvidia.com/cuda/doc/index.html`

Richard Bellman (1958), Lester Randolph Ford (1956), *Bellman–Ford algorithm*,
`https://en.wikipedia.org/wiki/Bellman-Ford_algorithm`

Moreno Marzolla (2024), *High Perfomance Computing*,
`https://www.moreno.marzolla.name/teaching/HPC/`

Microsoft (2024), *Windows API Index*,
`https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list`

AWS Docs (2024), *What's the Difference Between GPUs and CPUs?*,
`https://aws.amazon.com/compare/the-difference-between-gpus-cpus/`