Parallel Bellman-Ford in OpenMP and CUDA
Architectures and Platforms for Artificial Intelligence

Riccardo Murgia
Master's Degree in Artificial Intelligence, University of Bologna
riccardo.murgia2@studio.unibo.it

Abstract

This report presents a comprehensive analysis of parallel implementations of the Bellman-Ford algorithm, which solves the single-source shortest path problem on weighted graphs. The study explores the algorithm's performance when implemented using two parallel programming paradigms: OpenMP and CUDA C. The objective is to evaluate and compare the strong efficiency, weak efficiency, speedup, and other performance indices of these implementations. Multiple parallel versions of the algorithm and Various test scenarios have been implemented to emphasize different aspects of the algorithm's efficiency and scalability, including tests on graph size, edge density, and computational workload distribution.

1 Introduction

The Bellman-Ford algorithm is a method used to find the shortest paths from a source node to all other nodes in a graph, which can contain negative weights. This algorithm is particularly useful because, unlike Dijkstra's algorithm, it can handle graphs with negative weight cycles. The algorithm proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. If after $V - 1$ relaxations a shorter distance is found between the source vertex and any of the target vertex, then a negative cycle exists in the graph. The time complexity of the algorithm is $O(VE)$ where V is the number of vertices and E is the number of edges in the graph. The performance of the various algorithm versions has been tested on graphs of varying sizes with random weights within a specified range, defined by a minimum (lower bound) and a maximum (upper bound) value. The graphs were intentionally designed to be fully connected, representing the worst-case scenario characterized by $E = V^2$ edges. This deliberate choice allowed to analyze the algorithm's performance under maximum workload conditions. While ensuring the maximum workload, this approach does not compromise the generality of our study. The proposed solutions remain effective for graphs that are not fully connected. In such cases, where not all nodes are directly reachable from each other, the weights of non-existent edges can be set to $+\infty$. This trick guarantees that the algorithm operates correctly across various graph structures.
To achieve this objective and test all versions of the algorithm, a custom Graph structure has been implemented. The definition of this structure is as follows:

```
typedef struct {
    int origin;
    int end;
    int weight;
} Edge;
typedef struct {
    int num_vertices;
    int num_edges;
    int *nodes;
    Edge *edges;
    int **adjacency_matrix;
    int maximum_weight;
} Graph;
```

It can be noted that the structure includes both an edge list and an adjacency matrix to cater to different algorithm versions. This dual representation ensures flexibility to meet various Bellman-Ford algorithm requirements.

A Sequential version (Sq) was implemented to serve as a sanity check for the correctness of the solutions.

Algorithm 1 Sequential Bellman-Ford

procedure BELLMANFORD(G, w, s)
  Initialize $d[v] \leftarrow \infty$ for all $v \in V(G)$
  $d[s] \leftarrow 0$
  for $i = 1$ to $|V(G)| - 1$ **do**
    for each edge $(u, v) \in E(G)$ **do**
      if $d[u] + w(u, v) < d[v]$ **then**
        $d[v] \leftarrow d[u] + w(u, v)$
        $p[v] \leftarrow u$
      end if
    end for
  end for
  for each edge $(u, v) \in E(G)$ **do**
    if $d[u] + w(u, v) < d[v]$ **then**
      error ▷ negative-weight cycle
    end if
  end for
end procedure

2 Paralell Version

Six different parallel versions of the Bellman-Ford algorithm have been implemented using OpenMP (V0, V0_1, V1, V1_1, V2, V2_1), and an additional four versions have been implemented using CUDA (cuda_V0, cuda_V0_1, cuda_V1, cuda_V1_1).
These ten versions differ in the strategy used to parallelize the algorithm and in their approach to thread management, opting to reuse instantiated threads rather than creating and destroying them. In general, the operations that were attempted to be performed in parallel for each implementation, and thus the key points that will be described in more detail, are the initialization of distances, the internal computation at the $n-1$ relaxations -organized with various strategies, and various levels of parallelism depending on the version- and the final check for the presence of negative loops.

2.1 OpenMp Versions

- **V0**: This version operates on the edge list similarly to the sequential one but introduces parallelization for distance initialization, the internal computation within each relaxation, and the operation for detecting negative cycles.
  Specifically, the distance initialization is performed by creating $|V|$ tasks, each of which handles a job that assigns the value 0 to the source node and the largest representable value to the other nodes. To avoid overflow issues, the largest representable value is reduced by the largest weight present in the graph. Each relaxation creates $|E|$ tasks, each of which handles an edge. However, the operation of updating the distances and the predecessor list is performed within a critical section. The detection of negative cycles is also parallelized by creating other $|E|$ tasks, each assigned the job of checking a single edge. This process is implemented using a reduction mechanism on a sum. Each time the tasks are created also the T threads are created where T can be chosen based on the user preferences.

- **V0_1**: This version operates in the same way as the V0 version the only difference is that the parallelization parts are implemented creating the threads only once and reusing them for all the operations.

- **V1**: The implementation of this version is proposed in (Hua, December 23, 2020), in particular, Algorithm 2 contains the pseudo-code of the described logic. It parallelizes the distance initialization in the same way as version V0 and employs a single level of parallelization to perform the individual relaxations. The algorithm achieves parallelization by dividing the iteration over all edges in the graph into $|V|$ chunks, each containing all edges targeting the same destination vertex. Specifically, this version operates using the adjacency matrix and creates $|V|$ tasks, each responsible for processing all edges that share the same destination. Each task generates a list of all possible path's cost (candidate distances) of max length i that could reach node v with a better cost than the current best path of max length $i-1$ (stored in d).
  Subsequently, from this list, the candidate path of max length i that allows reaching v via the minor cost is selected by simply finding the minimum of the calculated costs and is compared to the current best path of length $i-1$. If the former is better than the latter then this new cost is considered as the shortest path.
  In the end, to detect negative cycles it has been used the same strategy as version V0, namely the algorithm creates other $|V|$ tasks to check if it is still possible to update the distances.
  This step ensures that no further updates can be made. Each time the tasks are created also T threads are created where T can be chosen based on the user preferences.

Algorithm 2 Parallel Bellman-Ford V1

```
1:  procedure BellmanFord(G, w, s)
2:      Init d[v] ← ∞ for all v ∈ V(G)
3:      d[s] ← 0
4:      Init p[v] ← None for all v ∈ V(G)
5:      Init new_d[v] ← None for all v ∈ V(G)
6:      for i = 1 to |V(G)| − 1 do
7:          for each v ∈ V(G) do                    ▷ Parallel
8:              for each u ∈ V(G) do
9:                  cand_d[u] ← d[u] + w(u, v)
10:             end for
11:             min_cand_dist ← min(cand_d)
12:             if min_cand_dist < d[v]) then
13:                 new_d[v] ← min_cand_dist
14:                 p[v] ← u
15:             else
16:                 new_d[v] ← d[v];
17:             end if
18:         end for
19:         swap(d, new_d)
20:     end for
21:     for each v ∈ V(G) do                        ▷ Parallel
22:         for each u ∈ V(G) do
23:             if d[u] + w(u, v) < d[v] then
24:                 error          ▷ negative-weight cycle
25:             end if
26:         end for
27:     end for
28: end procedure
```

- **V1_1**: This version operates in the same way as the V1 version the only difference is that the parallelization parts are implemented creating the threads only once and reusing them for all the operations.

- **V2**: This version implements the same logic used in version V1 but uses two levels of parallelization as presented in (Hua, December 23, 2020). The first-level parallelization is the same as the one in V1. The algorithm is parallelized at an additional level to the creation of the path's cost candidates and to the minimum-finding step, operations that in version V1 are made sequentially(rows number 8 and 11 of Algorithm 2). This second layer of parallelization is nested in the first one, which means that each task creates other additional V tasks and T threads. The V2 finds the minimum by implementing a custom reduction.

- **V2_1**: This version operates in the same way as the V2 version the only difference is that the parallelization parts are implemented creating the first level threads only once and reusing them for all the operations. The second-level threads are created and destroyed for every relaxation.

2.2 CUDA Versions

All CUDA-implemented versions required working with data stored in the GPU's RAM, so utilities were developed to copy previously defined graph struct. These utilities copy the entire structure, despite individual versions working with either the edge list or the adjacency matrix. In these implementations, the number of threads per block is fixed based on user-defined preferences, which then determines the number of blocks required to process all the operations. This calculation in version cuda_V0 (cuda_V1) is achieved by dividing the total number of edges (vertex) in the graph by the desired number of threads per block and taking the ceiling of the number. However, in versions cuda_V0_1 and cuda_V1_1, the calculation is more complex and will be explained subsequently.

- **cuda_V0**: This version is the CUDA implementation of the previously described V0 version. Specifically, it operates on the list of edges and utilizes three CUDA kernel functions: one for initializing distances, another for performing individual relaxations, and a final one for checking the presence of negative cycles. The first kernel function utilizes V threads to initialize the distances in the graph in the standard manner. The second one executes the i-relaxation and is implemented to guarantee that E threads process each one arc meaning that each thread has the job to update the distance of the destination node if needed. Updating the distances and the predecessor list, similar to version V0, is a critical part because some of the instantiated threads can try to access the same variable at the same time. This aspect is implemented using atomic operations to ensure correct and efficient updates. The final kernel function executed by E threads checks for negative cycles by utilizing shared memory to understand if the thread inside a block has detected a negative cycle at the end each master thread of each block writes on a global variable if needed using an atomic operation, which enhances performance and ensures accurate detection.

- **cuda_V0_1**: This version follows the same logic as the cuda_V0 version. It uses the same initialization procedure and negative cycle verification mechanism. However, it introduces a different kernel to process the relaxations, thereby avoiding the instantiation of new threads for each relaxation. The iteration through the $|V|-1$ relaxations is handled inside the kernel function, with an additional custom synchronization method between blocks. Specifically, to develop this mechanism, it was necessary to determine the optimal number of blocks (B) required to process the entire set of edges and the number of edges each thread has to process (G). A crucial aspect is that B must be less than or equal to the number of available multiprocessors to avoid deadlock. As told before the idea is to implement synchronization among blocks, it is important to ensure that no more blocks are instantiated than the available multiprocessors because if this were to occur, some blocks would find themselves waiting for a multiprocessor to become available. The multiprocessors though are already occupied with executing the assigned blocks, which would eventually reach a synchronization barrier between blocks, therefore resulting in a deadlock situation. As the group size increases, the total number of threads to be instantiated is calculated. Based on this, the total number of blocks is defined as:

$$\mathrm{B(G,\ T)} = \left\lceil \frac{\lceil \frac{E}{G} \rceil}{T} \right\rceil$$

  - E is the total number of edges.
  - G is the group size.
  - T is the number of threads per block.

  The goal is to have the smallest possible group size that meets the condition related to the number of blocks. This ensures efficient kernel execution, optimal use of the GPU, maximizing parallelism, and minimizing idle threads.
  In particular, as described, the synchronization mechanism between blocks uses a custom strategy. The first thread of each block serves as a signaler, informing a master thread when its block completes its assigned task. The master thread waits until all blocks have finished their work before updating a semaphore to start a new iteration. The other threads, once they have finished their work, wait for the semaphore to change before proceeding to a new relaxation. This mechanism adds some complexity but ensures proper synchronization and coordinated execution across all blocks.

- **cuda_V1**: This version uses the same initialization step used in version cuda_V0 and implements in CUDA the same logic used by the V1 version to perform the single relaxation. Specifically, it operates on the adjacency matrix and like the other CUDA version utilizes three CUDA kernel functions: one for initializing distances, another for performing individual relaxations, and a final one for checking the presence of negative cycles.

- **cuda_V1_1**: This version operates in the same way as the cuda_V1 version. It uses the same initialization procedure and negative cycle verification mechanism. The only difference is that the parallelization of the relaxation steps is implemented like in version cuda_V0_1 using the synchronization between block mechanism.

3 Experimental setup

To evaluate the characteristics of each implementation, four tests were conducted, each selectable by activating a specific flag when launching the executable:

- **Development Phase Test:**
  The first test was used exclusively during the development phase and is currently disabled in the project's final presentation. This test allowed for verifying the correctness of the various implementations. Specifically, this test, selectable by setting the first flag to 1, enables the manual insertion of certain parameters to generate graphs of arbitrary size with edges defined by an arbitrary interval [lower bound, upper bound]. It also allows specifying the number of tests to be performed, displaying the lists of edges, adjacency matrices, and solutions produced by each algorithm with corresponding execution times, and, in the case of multiple tests, the mean value and the standard deviation of execution times for each algorithm.
  It is important to note that in this test, as well as in the others, the graphs are generated randomly based on incremental seeds according to the number of tests to be executed. This approach ensures the reproducibility of the results.

- **Strong Efficiency Test:**
  The second test, executable setting up the second flag to 1, was implemented to assess the executioner time varying the number of threads used, the strong efficiency, and the speed-up of the OpenMP versions. Specifically, this test runs the OpenMP versions on graphs with 1000 vertices, varying the number of threads used from 1 to a maximum of 8. For each configuration, ten tests are conducted each using a different random graph to ensure the reliability of the experiment. Therefore execution times are collected and means and standard deviations are computed.

- **Weak Efficiency Test:**
  The third test, executable setting up the third flag to 1, evaluated the weak efficiency of the OpenMP versions by executing the algorithms with varying workloads. Specifically, it started with graphs of 1000 nodes and incremented their size by a factor of $\sqrt[3]{k}$, with k varied from 1 to 8. This factor was also used to increase the number of threads used in the execution. Similar to the Strong Efficiency test, for each configuration, three tests are conducted, each using a different random graph.

- **OnpenMP VS CUDA Test:**
  The fourth and last test, executable by setting the fourth flag to 1, collected the execution times for each version to process random graphs of sizes 100, 500, 1000, 2000, 5000, and 10000. For this test, the number of threads for the OpenMP versions was fixed at 4, while for the CUDA versions, the number of threads per block was set to 1024, following the specifications of the provided devices. It should also be noted that for each fixed graph size, three tests were conducted, calculating the averages and standard deviations of the execution times as done in the other tests.

All the tests described above are aimed at collecting execution times, which are exported to specific text files. These text files are then imported in a .ipynb file that computes the speed-up, the strong efficiency indexes, the weak efficiency indexes, the throughput, and the percentage of time that the CUDA versions spend copying the structure representing the graph to the GPU. The script also generates graphs to illustrate the performance metrics, making it easier to visualize and analyze the results.

4 Results

The execution times summarized in Figure 1, show the performance trends of the six algorithms parallelized using OpenMP as the number of threads utilized varies on a fully-connected graph of 1000 nodes. Specifically, the plot highlights that versions V1, V1_1, V2, and V2_1, when executed using a single core, take significantly more time compared to the sequential algorithm due to the additional operations required for parallelization. Conversely, versions V0 and V0_1 exhibit performance similar to the sequential version when using a single core.
The figure also highlights that the thread reuse strategy, applied in versions V0_1, V1_1, and V2_1, does not result in a significant performance improvement compared to their respective counterparts that destroy and recreate threads at each relaxation.
Additionally, it can be observed that algorithms V1, V1_1, V2, and V2_1 achieve the shortest execution time with three cores.
Versions V0 and V0_1 also achieve their best execution time with three cores, with execution times that are very

similar to those achieved with four cores.
These results suggest that while some versions benefit from parallelization, others suffer from significant overhead when the number of cores is limited.

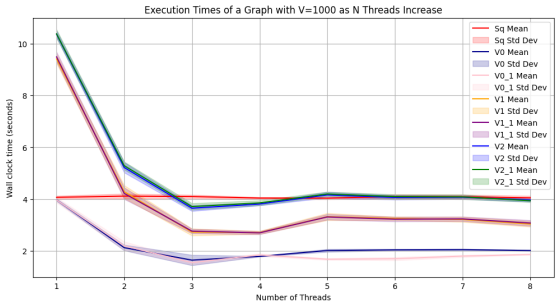

Figure 1

The plots in Figure 2 respectively depict the speedup and strong efficiency indices associated with the OpenMP versions. Specifically, it's observed that versions V1 and V1_1, when executed with 4 threads, significantly reduce the execution time compared to sequential execution. However, the individual efficiency of each thread is not optimal, as highlighted by the fact that the overall efficiency index reaches its peak with 3 threads. This suggests that these algorithms benefit from parallelism up to a certain point, beyond which the overhead of thread management starts to negatively impact the ideal performance. Therefore, using 3 threads would allow the algorithms to achieve a better balance between parallelism and management overhead, even though the total execution time isn't minimized.
Conversely, versions V2 and V2_1 achieve the highest speedup factor and the highest strong efficiency index when executed with 3 threads. On the other hand, versions V0 and V0_1 reach their maximum speedup factor when executed with 3 threads, but differ in the strong efficiency index. Specifically, version V0 reaches its maximum when executed with 2 threads, while version V0_1 reaches its best performance in strong efficiency when executed with 3 threads.

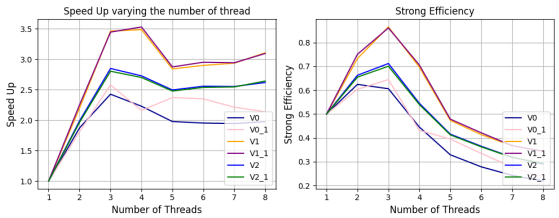

Figure 2

The plot in Figure 3, depicting the trend of weak efficiency for the algorithms implemented with OpenMP, shows that the algorithms generally exhibit decreasing weak efficiency with increasing k. They maintain good efficiency up to $k = 4$ but experience a significant drop for higher values of 4. This behavior indicates the non-optimal scalability of the algorithms on the processor used for the tests.

Figure 3

The graphs in Figure 4 present a comparison of the execution times of various implemented versions. Specifically, they compare different implementations of the same logic using different paradigms. These graphs provide further confirmation that thread reuse strategies do not bring significant improvements compared to the counterpart that creates and destroys threads at each relaxation.
As expected, the CUDA versions are the most performant on large graphs. However, as shown in the graph in Figure 5, they are penalized by the operations of copying the graph to the GPU's RAM. Finally, it can be observed that versions V2 and V2_1, which use double-level parallelism, are not particularly performant. In fact, increasing the number of threads used in the second level of parallelism excessively burdens the overall performance of these versions, leading them to become slower than the sequential version, emphasizing the non-optimal management of nested parallelism. Therefore, they were tested using a single thread. Using a single thread though transforms these versions to be similar to versions V1 and V1_1 with the only difference being that they perform the minimum path search through a reduction mechanism with just two threads (The master and the slave created at each iteration).

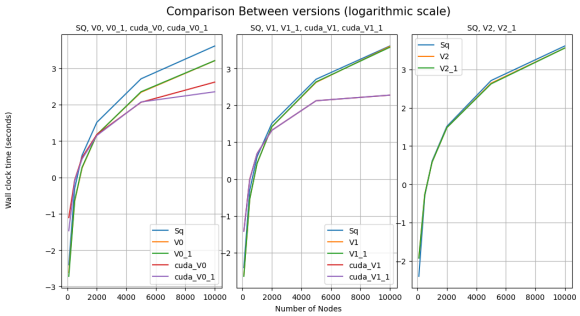

Figure 4

As mentioned earlier, the graph in Figure 5 focuses on the time required to copy the graph to the GPU's RAM. Specifically, it highlights how this factor becomes increasingly significant in total execution times as the graph size decreases. It's worth noting that this aspect is accentuated by the use of generic copy utilities not specific to each version. Both the cuda_V0 (cuda_V0_1) and cuda_V1 (cuda_V1_1)

versions, which respectively operate using the edge list and the adjacency matrix, share a generic GPU copy mechanism. This mechanism handles transferring the entire graph represented by the previously defined data structures, even though this means transferring both the adjacency matrix and the edge list.

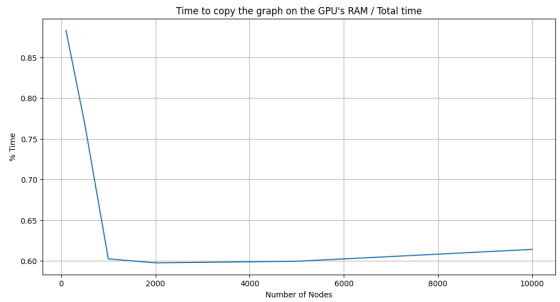

Figure 5

In conclusion, the plot depicted in Figure 6 illustrates the computational speed, measured in bytes per second, at which the various implemented versions are capable of processing data as the graph size increases. Specifically, it is evident that the CUDA versions exhibit notably higher speeds compared to the versions implemented in OpenMP

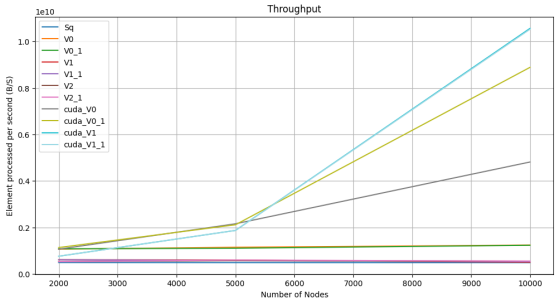

Figure 6

5 Future Work

Several potential avenues for future work have been identified to enhance the performance and capabilities of the current implementations:

- **Recycling Second-Level Threads in Versions V2 and V2_1**: Developing a mechanism to recycle the second-level threads in versions V2 and V2_1 could be explored. This would allow us to verify if increasing the number of threads used in the second level of parallelism leads to performance improvements.
- **CUDA Implementation for Versions V2 and V2_1**: Another promising direction is to develop the logic of versions V2 and V2_1 using CUDA. This could potentially leverage the strengths of GPU parallelism for these more complex versions.
- **Optimizing Data Transfers in CUDA Versions**: To improve the performance of the CUDA versions, it would be beneficial to minimize the elements copied to the GPU. This optimization aims to reduce the significant impact of copy time on overall execution time for smaller graphs.
- **Optimizing Minimum Path Search**: Further optimization could be achieved by improving the management of the minimum path search among candidate distances. The current method, as described, involves storing all possible candidate path costs. Implementing an on-the-fly minimum search mechanism could avoid the need to store all candidate distances, thus potentially enhancing performance.

6 Links to external resources

The repository with the code can be found at: https://github.com/RiccardoMurgia/Parallel_Bellman_Ford_Implementation

References

Zhe Hua. December 23, 2020. Coms 4995: Parallel functional programming parallel bellman-ford algorithm.