Architectures and Platforms for Artificial Intelligence, Module 1

Daniele Napolitano daniele.napolitano4@studio.unibo.it

February, 2023

1 Introduction

The **single-source shortest path (SSSP)** problem is one of the fundamental problems in graph theory. It asks for finding the shortest paths from a given source vertex to all other vertices in a weighted graph.
The Bellman-Ford algorithm is a well-known solution to this problem, which works by relaxing the edges of the graph iteratively until the optimal distances are found.
However, the sequential version of the Bellman-Ford algorithm has a time complexity of $O(|V||E|)$, where $|V|$ and $|E|$ are the number of vertices and edges in the graph, respectively. This makes it impractical for large-scale graphs, therefore parallelizing the Bellman-Ford algorithm can potentially improve its performance and scalability.
In this work, I experimented with parallel programming by implementing a parallel version of the Bellman-Ford algorithm comparing two different approaches:

- **OpenMP** is a widely used API for shared-memory parallel programming, which allows us to use multiple threads to execute the algorithm on a multicore CPU.
- **CUDA** is a language extension for general-purpose GPU programming, which exploit the massive parallelism of NVIDIA's devices.

The results obtained with these methods will be evaluated individually and compared, also analyzing their ability to scale on very large graphs.

1.1 Bellman-Ford sequential algorithm

The pseudocode to define this algorithm is shown in Algorithm 1, and was originally taken from [1].

Algorithm 1 Bellman-Ford pseudocode

```
1:  function BELLMANFORD($G, S$)
2:      for each vertex $V$ in $G$ do                        ▷ Initialization phase
3:          $distance[V] \leftarrow \infty$
4:          $previous[V] \leftarrow$ NULL
5:      $distance[S] \leftarrow 0$
6:
7:      for each vertex $V$ in $G$ do                        ▷ Relaxation phase
8:          for each edge $(U, V)$ in $G$ do
9:              $tempDistance \leftarrow distance[U] + edge\_weight(U, V)$
10:             if $tempDistance < distance[V]$ then
11:                 $distance[V] \leftarrow tempDistance$
12:                 $previous[V] \leftarrow U$
13:
14:     for each edge $(U, V)$ in $G$ do                     ▷ Negative cycles detection
15:         if $distance[U] + edge\_weight(U, V) < distance[V]$ then
16:             Error: Negative Cycle Exists
17:     return $distance[], previous[]$
```

The source node will be always set as the first one (i.e. index zero).
The relaxation step is the most expensive, as it requires to go through all the edges of the graph. For both OpenMP and CUDA approaches, the most amount of effort will go into parallelizing that section in order to make it more efficient and faster to run.
The program *sequential_version.c* implements this pseudocode, but it is used only for debugging, to check that the results of the parallel versions are correct, and it is not used to compute sequential times.

1.2 Graph generator

To test the scalability of the parallel algorithms, there is a need of huge graphs with thousands of vertices to test with.
Since it was hard to find such instances, I managed to create a random graph generator, modifying an already existing one from [2].
Given any number of vertices, and a maximum number of edges for each vertex, it generates a graph with the following characteristics:

- Positive weights values in range $[1, 50]$.
- Directed, without self cycles.
- Not fully connected.
- The maximum number of edges for each vertex is half the number of total vertices (new edges are generated with a probability of 0.5).
- Fixed random seed equal to 42.

2 OpenMP version

The main directive used to parallelize the algorithm was the **pragma omp for**: by splitting the loops of the for loops and assigning them to different threads, the overall speed has the potential to highly increase for large graphs (with many edges).
It was implemented in:

1. **Initialization** loop.
2. Inner loop of the **relaxation** phase (loop over the edges).
3. **Negative cycles detection** loop.

A *static* schedule was chosen for all the phases, since because the operations have the same level of complexity, moreover, the *dynamic* one performed poorly in the experiments.

2.1 Relaxation phase

For the relaxation phase, it was possible to only parallelize the inner loop on the edges, because there is a data dependency between the different iterations of the outer vertex loop: it is in fact a "timestep" loop that updates at every step the distance and previous vectors, and future timesteps could depend on the values computed at the previous cycle.

3 CUDA version

The CUDA version was developed with the same idea of the OMP one: The inner loop of the relaxation phase was parallelized with a call to a device function:

```
relax<<<(tE + BLKDIM-1) / BLKDIM, BLKDIM>>>(edges, d, p, tE);
```

Given a block size of 256 threads, the block count is set to $\frac{tE+255}{256}$ to **assign each edge to a thread** (tE is the total number of Edges).
A similar thing is done for the initialization phase, although using the number of vectors (tV) :

```
initialize<<<(tV + BLKDIM-1) / BLKDIM, BLKDIM>>>(d, p, tV, source);
```

And the negative cycle detection loop:

```
checkNegativeCycles<<<(tE+BLKDIM-1)/BLKDIM,BLKDIM>>>(edges, d, tE);
```

Inside the relative *__device__* functions, the index of the edge or vertex associated to the thread, is computed in the same way:

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
```

4 Evaluation

To evaluate performances and efficiency for the two implentations, several experiments were performed with different graphs and number of threads, to understand the strenghts and weaknesses of each method.

4.1 Speedup analysis

For both CUDA and OMP versions, the T_{serial} argument for the speedup was computed using the same parallel code, with only one thread (for both) and one block (for CUDA), to avoid spurious superlinear speedup phenomena that could arise from cache effects or algorithmic differences.
To test the scalability of the two implementations, six graphs were generated with an increasing number of vertices: 10, 50, 100, 250, 500, 750, 1.000 and 5.000. The relative number of edges grows exponentially.
All the time measures are measured and presented in wall-clock time, checking the duration of the whole *bellmanFord* function, thus excluding input and output times, and other initialization operations in the main function which are not relevant for the algorithm.
Table 1 shows all the execution times for both implementations, and their respective sequential version (using a single thread and block). *N/A* values refers to time of execution greater than a time limit of 5 minutes (i.e. 300 seconds). The same values are also plotted in Figures 1 and 2.

		CUDA			**OMP**		
vertices	edges	$T_{parallel}$	$T_{sequential}$	$S(p)$	$T_{parallel}$	$T_{sequential}$	$S(p)$
10	24	0.0862	0.0874	1.01	0.0010	0.00001	**0.01**
50	658	0.0934	0.0895	0.96	0.0040	0.0003	**0.06**
100	2 523	0.0952	0.1223	1.28	0.0074	0.0012	**0.16**
250	15 594	0.0931	0.6032	6.48	0.0268	0.0165	**0.62**
500	62 490	0.0862	4.3474	50.46	0.1038	0.1320	1.27
750	140 728	0.0915	14.4331	157.68	0.2944	0.4422	1.50
1000	249 562	0.1588	33.1761	208.96	0.6568	1.0422	1.59
5000	6 250 107	0.7717	N/A	N/A	41.6592	131.4383	3.16

Table 1: Wall Clock times and Speedup for each implementation. OMP used 8 threads for $T_{parallel}$.

In Table 1, the linear speedup is computed as $Speedup(p) = \frac{T_s}{T_p} = \frac{T_1}{T_p}$, and is considered as an approximation of the real speedup, which should be computed using Amdahl's law.
CUDA has less syncronization overheads for smaller instances, but runs generally slower than OpenMP for the same ones. It gets much faster for bigger instances, starting from the 500 vertices graph, but the serial version is very slow compared to its OpenMP counterpart.

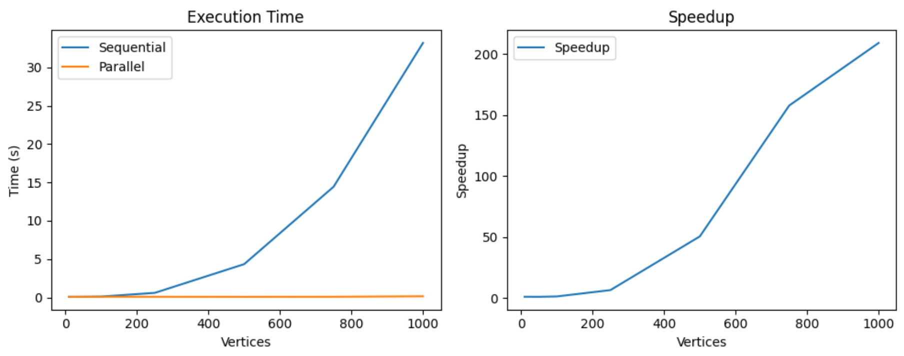


Figure 1: Times and speedup for CUDA. Does not include the 5000 vertices instance.

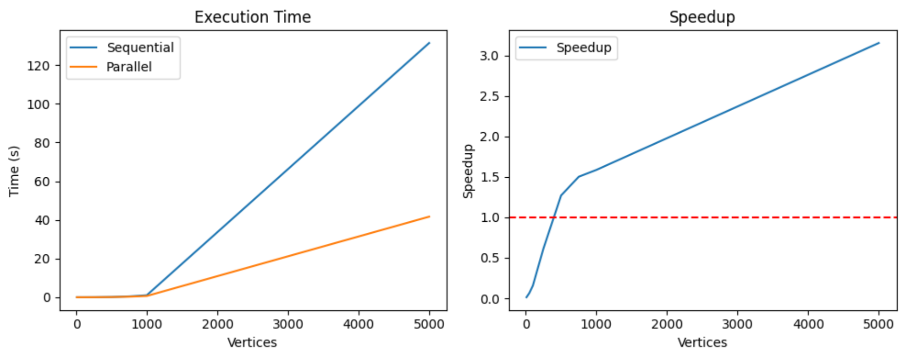


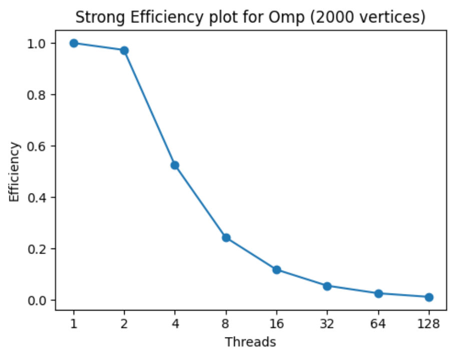
Figure 2: Times and speedup for OpenMP

4.2 Strong Scaling Efficiency for OpenMP

For the OpenMP variant, the number of threads is just a parameter, and the assignment of threads is automatically managed by the system when running the *omp parallel for* directive. For this reason, it is possible to test different numbers of threads on the same input graph. The speedup values $S(p) = \frac{T(1)}{T(p)}$ were computed for p=2,4,8,16,32,64,128 with a fixed graph size of 2 000 vertices (1 000 399 edges), and plotted in Figure 3, along with the efficiency values: $E(p) = \frac{S(p)}{p}$.

It is evident that the speedup rises up until 4 threads, then slowly decreases. This behaviour agrees with Amdhal's law, stating that the speedup has an upper limit, given by the serial fraction of the algorithm (not parallelizable). The sweet spot with good balance between performance and low overhead for synchronization lies around 4 threads.

The strong scaling efficiency score greatly decreases soon after 2 threads, but



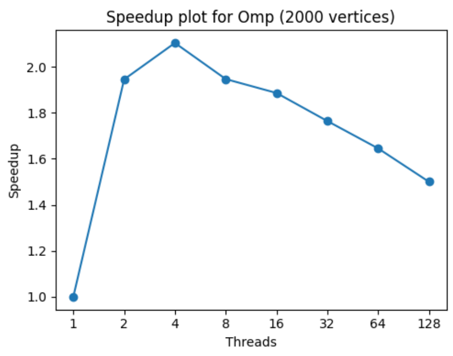


Figure 3: Strong Efficiency (left) and Speedup (right).

this might be linked to the number of physical cores of the CPU of the SLURM environment.
This kind of analysis is not doable for the CUDA version, since it is coded in such a way that the number of maximum threads (for the relaxation phase), equals to the number of edges.

5 Conclusion

After trying out and evaluating both versions, it is evident that CUDA achieves a much higher speedup than OpenMP, as it takes advantage of the massive SIMD parallelism of the GPU. Furthermore, since the latter only uses the CPU, it is constrained to a smaller number of usable parallel threads, and shows a greater synchronization overhead.
Despite the single process version of CUDA ($p = 1$) being unable to handle larger instances, rendering speedup calculations impossible, the parallel version exhibited impressive performance on graphs with millions of vertices. For instance, the 6M+ graph (5 000 vertices), achieved a wall clock time of less than one second (see Table 1).

References

[1] Bellman ford's algorithm. Available at `https://www.programiz.com/dsa/bellman-ford-algorithm#google_vignette`.

[2] Manish Bhojasia. C program to create a random graph using random edge generation. Available at `https://www.sanfoundry.com/c-program-generate-random-undirected-graph-given-number-edges/`.