Parallelization of the Floyd-Warshall algorithm

David Bertoldi – 735213
email: d.bertoldi@campus.unimib.it

Department of Informatics, Systems and Communication

University of Milano-Bicocca

Abstract—The well known Floyd-Warshall (FW) algorithm solves the all-pairs shortest path problem on directed graphs. In this work we parallelize the FW using three different programming environments, namely MPI, OpenMP and CUDA. We experimented with multiple data sizes, in order to gain insight on the execution behavior of the parallelized algorithms on modern multicore and distributed platforms, and on the programmability of the aforementioned environments. We were able to significantly accelerate FW performance utilizing the full capacity provided by the architectures used.

1 Introduction and Background

The FW is a classic dynamic programming algorithm that solves the *all-pairs shortest path (APSP)* problem on directed weighted graphs $G(V, E, w)$, where $V = \{1, \dots, n\}$ is a set of nodes, $E \subseteq V \times V$ are the edges and w is a weight function $E \to \mathbb{R}$ that expresses the cost of traversing two nodes. The number of nodes is denoted by n and the number of edges by m .

The output of the algorithm is typically in matrix form: the entry in the ith row and jth column is the weight of the shortest path between nodes i and j. FW runs in $\Theta(|V|^3)$ time and for this reason is a good choiche when working with dense graph: even though there may be up to $\Omega(|E|^2)$ edges, the computational time is independent from the number of edges.

The FW algorithm is shown in Alg. **??**

A C implementation of this algorithm can be found here; it is also referred in this document as *sequential* implementation and it is used as base version when comparing to parallel implementations.

In particular we define the *speedup* (S) of a given version v of the algorithm as it follows:

$$S = \frac{T_s}{T_v}$$

where T_v is the execution time of the version v of FW and T_s is the execution time of the *sequential* version.

It is easy to notice that that the nested i and j for-loops are totally independent and therefore parallelizable.

Algorithm 1: The Floyd-Warshall (FW) algorithm

```
for $(u, v) \in E$ do
    $M_{u,v} \leftarrow w(u, v)$
end
for $v = 1 \to n$ do
    $M_{v,v} \leftarrow 0$
end
for $k = 1 \to n$ do
    for $i = 1 \to n$ do
        for $j = 1 \to n$ do
            if $M_{i,j} > M_{i,k} + M_{k,j}$ then
                $M_{i,j} \leftarrow M_{i,k} + M_{k,j}$
            end
        end
    end
end
```

2 Parallel Programming Environments

3 Computational Platforms and Software Libraries

This section describes technical details about the implementation of the solutions exposed in this document. It also contains the collected data under analysis.

3.1 Sequential implementation

The *sequential* version of the FW algorithm can be found here. The program is compiled as it follows:

```
$ gcc sequential.c -o sequential.out -O3
```

notice the `-O3` flag that makes the program run 3.5 times faster. The program accepts 2 arguments:

```
$ ./sequential.out <v> <d>
```

where `v` is the number of verteces, expressed ad positive integer, and `d` is the density of the presence of edges, expressed as an integer from 0 to 100.

The `gcc` version used for this work is 7.5.0 and the program ran on a Intel Core i7-9700K.
Table 1 shows the execution time (expressed in milliseconds) depending on the number of vertices.

Vertices	Execution time (ms)
1000	1095
2000	8860
5000	138643
7500	468750
10000	1112111
12500	2170138

Table 1: Execution time of the *sequential* FW

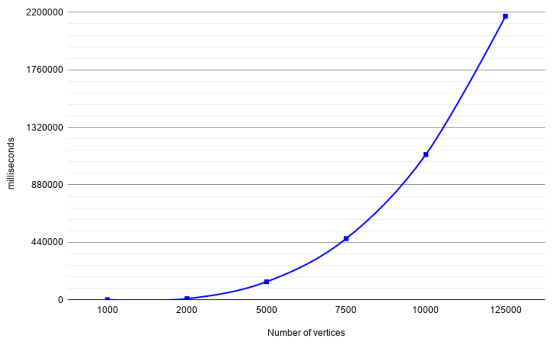

Figure 1: Trend of the execution time based on Table 1

Figure 1 shows the trend of the execution time.

It is easy to notice that the graph represents a third grade curve; this is the interpolated function starting from the collected data:

$$f(n) = 2.22n^3 - 18.83n^2 + 92.14n - 94.84 \approx \Theta(n^3)$$

3.2 MPI implementation

The *MPI* version of the FW algorithm can be found here. The program is compiled as it follows:

```
$ mpicc -g -Wall mpi.c -o mpi.out -O3
```

Like the *sequential* version, the program accepts 2 arguments:

```
$ mpirun -np <p> mpi.out <v> <d>
```

The `MPI` version used for this work is 2.1.1 and the program ran on a cluster of 8 nodes over a LAN.
Table 2 shows the execution time (expressed in milliseconds) and the percentage of time spent in initialization and communication, depending on the number of processors; the program computes the *APSP* problem for 5040 vertices.

Processors	Execution time (ms)	MPI %
1	90842	0.06
2	46608	2.57
3	30718	4.45
4	24457	5.47
5	20200	6.95
6	17403	8.35
7	15563	9.12
8	13812	10.1

Table 2: Execution time of the *MPI* FW

3.3 OpenMP implementation

The *MPI* version of the FW algorithm can be found here. The program is compiled as it follows:

```
$ g++ -fopenmp openmp.c -o openmp.out -O3
```

Unlike the *sequential* version, the program accepts 3 arguments:

```
$ ./openmp.out <v> <d> <t>
```

where `t` is the number of threads OpenMP can use for parallelization.
The `gcc` version used for this work is 7.5.0 and the program ran on a Intel Core i7-9700K, which has 8 core with no Hyper-Threading. Table 3 shows the execution time (expressed in milliseconds) depending on the number of vertices and available cores.

Vertices	x2 threads (ms)	x4 threads (ms)	x8 threads (ms)
1000	541	364	158
2000	4365	2921	1260
5000	69590	46443	19496
7500	230260	155531	65118
10000	543900	367434	153682
12500	1063129	716363	299006

Table 3: Execution time of the *OpenMP* FW

3.4 CUDA implementation