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 a directed weighted graph $G(V, E, w)$, where $V = \{1, \ldots, n\}$ is a set of nodes, $E \subseteq V \times V$ are the edges and w is a weight function $E \rightarrow \mathbb{R}$ that expresses the cost of crossing 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 i^{th} row and j^{th} 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 **Algorithm 1**.

A C implementation of this algorithm can be found here; this version is referred in this document as *serial* FW and it is used as base for comparison.

Algorithm 1: The Floyd-Warshall (FW) algorithm

```
1  for $(u, v) \in E$ do
2      $M_{u,v} \leftarrow w(u, v)$
3  end
4  for $v = 1 \rightarrow n$ do
5      $M_{v,v} \leftarrow 0$
6  end
7  for $k = 1 \rightarrow n$ do
8      for $i = 1 \rightarrow n$ do
9          for $j = 1 \rightarrow n$ do
10             if $M_{i,j} > M_{i,k} + M_{k,j}$ then
11                 $M_{i,j} \leftarrow M_{i,k} + M_{k,j}$
12             end
13         end
14     end
15 end
```

2 Terminology

In this document we use specific terms that refer to specific definitions. Such terms are *emphasized* in order to distingiush them from everyday terms and we listed them in this section, which acts as a dictionary.

Edge/cell under analysis

The edge/cell in the distance matrix selected by FW for a given i and j. Often noted as $M_{i,j}$ within this document.

Intermediate edges

The two edges which sum must be compared with the edge *under analysis* (line 10 of **Algorithm 1**). They are noted as $M_{i,k}$ and $M_{k,j}$ within this document.

Serial FW

The implementation of FW as shown in **Algorithm 1**: the execution is performed on a single process/thread.

Speedup

For a given implementation v of FW, it is defined as follows:

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

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

Efficiency

For a given implementation v, it is defined as follows:

$$E_v = \frac{S_v}{p}$$

where S_v is the speedup of version v of FW and p is the number of processor units involved in the computation; the closer E_v remains to 1 while increasing p, the better.

3 Methodology

3.1 Strategy

It's easy to notice that the nested i and j for-loops in **Algorithm 1** can be parallizable, but one of the *intermediate edges* for a process could be the *edge under analysis* for another process. For example process P^1 is writing in a cell while process P^2 is reading from the same cell, leading to a unpredictable final result if P^1's writing comes before or after P^2's reading.

It seems that enabling parallelization requires some sort of blocking mechanism between processes, like a semaphore, but we prove that no data race can occur as in the previous example as far as the value of k is shared between processes.

Let's assume we have 2 processes, namely P^1 and P^2, and they have *under analisys* $M_{i,j}$ and $M_{x,y}$ respectively, with $x \neq i$ so that each process partitions the matrix horizontally. At any point the following system of preconditions exists (the two inequalities are taken from **Algorithm 1** at line 10):

$$\begin{cases} P^1_{i,j} & > & P^1_{i,k} & + & P^1_{k,j} \\ P^2_{x,y} & > & P^2_{x,k} & + & P^2_{k,y} \end{cases} \tag{1}$$

i.e. P^1 is reading values of $M_{i,k}$ and $M_{k,j}$ so it can decide if $M_{i,j}$ should be overwritten; the same does P^2 with $M_{x,k}$, $M_{k,y}$ and $M_{x,y}$ respectively.
In order to have a data race, the following statement must be true as well

$$(x = i \wedge k = j) \vee (k = i \wedge y = j)$$

i.e. P^2 is using as one of its *intermediate edges* the *edge under analysis* of P^1. Without losing of generality we do not analyze the opposite case because the proof develops symmetrically.
Because $x \neq i$, only the following must be verified

$$k = i \wedge y = j \tag{2}$$

and by applying (2) to (1) we have

$$P^1_{i,j} > P^1_{k,k} + P^1_{i,j} \tag{3}$$

but (3) is clearly false, because M is a hollow matrix *i.e.* the diagonal elements are all equal to 0, leaving the following inequality:

$$P^1_{i,j} > P^1_{i,j} \tag{4}$$

Clearly no number can be greater than itself and this means that at this point the comparison made by P^1 is irrelevant and is always evaluated to false. If P^2 writes after or before P^1's evaluation does not really matter and thus there is no data race as far as k is the same for all the running processes.

This is really important because it assures that there is no need for locking mechanism on M when it comes to parallelize the nested i and j for-loops and having no blocks guarantees better performance.

Now the strategy relies on dividing the matrix by rows and assigning an equal number of rows to each process. Once k is set and it's shared among all processes, each process puts *under analysis* every cell of its sub-matrix and selects its *intermediate edges* depending on k.

In this work we propose 3 parallel architectures implemented with 3 different parallel programming environments: *Distributed computing* with MPI, *Shared-memory Mutithreading* with OpenMP and *GPGPU* with CUDA.

The document analyzes each implementation and gives an overview of timings, implementations and pros and cons for each case.

3.2 Distributed with MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

The main strategy is based on scattering horizontally the whole matrix among all the process, so that each process can read a portion of the matrix of size $\frac{n^2}{p}$; then a *process of competence* is chosen: as k is in common (but never transmitted) to all the processes, there's always a cell in the k^{th} row representing one of the two intermediate vertices for any process and there's always one process which this row was assigned to. The value k is always "up to date" among processes because each for-loop involving k starts with a collective communication which implies a synchronization point among processes.

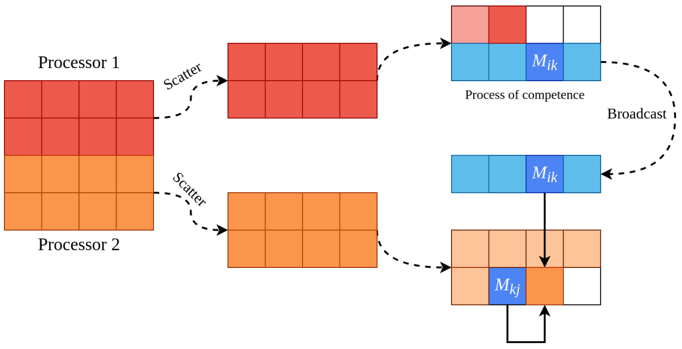

Figure 1: View of the data each thread can reach

Everytime k changes, the k^{th} row is broadcasted by the *process of competence* (the process that "owns" the row) to the other processes; a total of k `MPI_Bcast` operations are required.

Once the k^{th} row has been received, each process acts like the original FW in the i and j for-loops; obviously they each write values in their own local matrix.

At the end of the k for-loop, all the local matrices are gathered to the root process. **Algorithm 2** shows an high level pseudo-code of the strategy exposed above. For the concrete implementation see **section 4.2**.

We can list all the communication required:

Algorithm 2: Distributed version of FW

```
1  broadcast($n$, $ROOT$)
2  scatter($M$, $p$)
3  for $k = 1 \rightarrow n$ do
4      if $rank$ = findOwnerOfK$^{th}$Row$(k, rank)$ then
5          $K \leftarrow$ getRow($M$, $k$)
6      end
7      broadcast($K$)
8      for $i = 1 \rightarrow \frac{n}{p}$ do
9          for $j = 1 \rightarrow n$ do
10             if $M_{i,j} < M_{i,k} + K_j$ then
11                 $M_{i,j} \leftarrow M_{i,k} + K_j$
12             end
13         end
14     end
15 end
16 gather($M$, $p$);
```

- 1 `MPI_Bcast` for communicating the value of n
- 1 `MPI_Scatter` for the assignment of the local sub-matrix
- n `MPI_Bcast` for communicating the k^{th} row
- 1 `MPI_Gather` for the collection of the local sub-matrix

Table 1 approximates how many bytes are involved in the communications, assuming that the implementation uses 4 bytes to store one `int` and omitting the overhead of the communication protocol.

Count	Type	Size
1	`MPI_Bcast`	$4(p-1)$ bytes
1	`MPI_Scatter`	$4\frac{n^2(p-1)}{p}$ bytes
n	`MPI_Bcast`	$4n^2(p-1)$ bytes
1	`MPI_Gather`	$4\frac{n^2(p-1)}{p}$ bytes

Table 1: Approximation of the size of each communication for *MPI* FW

Total communications can be expressed with the following formula, that can help to calculate the bandwidth of the network required for this implementation:

$$W_{comm} = 4(p-1)(1 + n^2(1 + \frac{2}{p})) \text{ bytes}$$

This formula is important because the time spent on communications is time taken from the calculation and a network with an inadequate bandwith is the main source of bottlenecks. For example having 8 processes consuming a 12500×12500 matrix implies a total of ~5.46GB transferred over the entire network; this means that each processor would theorically spend 5468ms (43 seconds in total) on communication over a 1Gbps network.

Figure 2 shows the amount of time taken to initialize the MPI cluster and communicate during the execution of the *MPI* FW. In this case a dense 5040×5040 matrix is used as input and the MPI cluster was located in a completely-connected network with 1Gbps bandwith.

The bandwidth required has a quadratic and linear growth with respect to the number of vertices and the number of processes respectively; so with a medium-small matrix the time used in communication can exceed the 10% of the total computational time. But by increasing the size of the matrix to 12600×12600 the time used in communication drops to 4%, with a total of 50 seconds. Thus the implementation fits well for huge matrixes, rather than medium ones.

We can also easily calculate the time required by the algorithm with this formula:

$$T = t_c\frac{n^3}{p} + log(p) \cdot (t_s(3+n) + t_w(1 + 2\frac{n^2}{p} + n))$$

where t_c is the computation time, t_s the time required to start a communication and t_w the time to communicate a single word (in this case a `int`); the formula assumes that the collective communications use a tree-based hierarchical pattern that takes $log(p)$ steps rather than p steps for p processes.

For this reason the speedup is considerable but far from the ideal for a 5040×5040 matrix. **Figure 3** shows the trend of the speedup in relation to the sequential version.

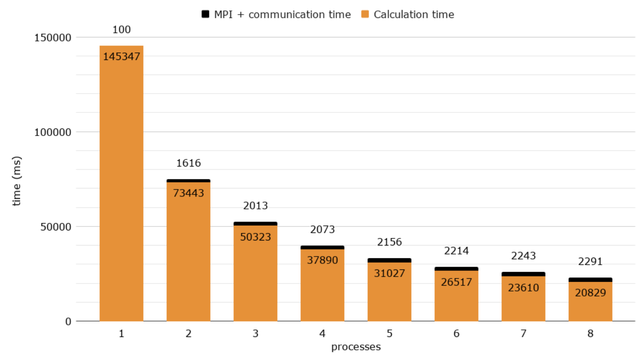

Figure 2: Calculation and MPI instantiation + communication time of *MPI* FW consuming a 5040×5040 matrix over a 1Gbps network depending on the number of processes

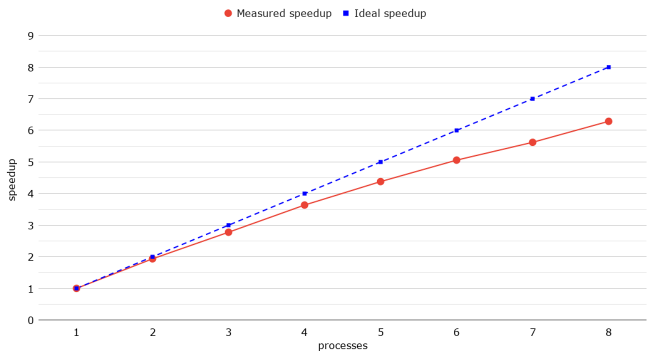

Figure 3: Speedup of *MPI* FW

With a consequent decrease in efficiency as shown in **Figure 4**

3.3 Shared-memory multiprocessing with OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) for parallel programming intended to work on shared-memory architectures. More specifically, it is a set of compiler directives, library routines and environmental variables, which influence runtime behavior. OpenMP

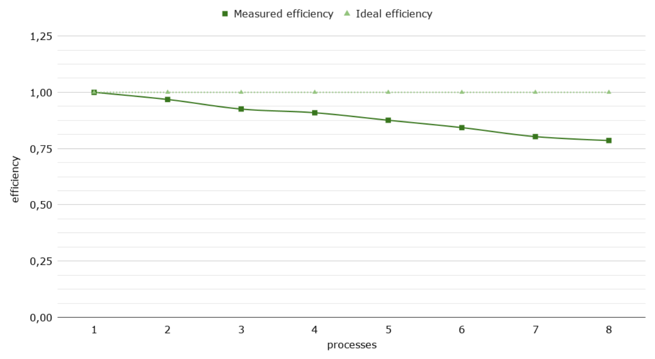

Figure 4: Efficiency of *MPI* FW consuming a medium matrix

enables parallel programming in various languages, such as C, C++ and FORTRAN and runs on mostoperating systems.

The OpenMP API uses the fork-join model of parallel execution. Multiple threads perform tasks defined implicitly or explicitly by OpenMP directives. All OpenMP applications begin as a single thread of execution, called the initial thread. The initial thread executes sequentially until it encounters a parallel construct. At that point, this thread creates a group of itself and zero or more additional threads and becomes the master thread of the new group. Each thread executes the commands included in the parallel region, and their execution may be differentiated, according to additional directives provided by the programmer. At the end of the parallel region, all threads are synchronized.

The runtime environment is responsible for effectively scheduling threads. Each thread, receives a unique id, which differentiates it during execution. Scheduling is performed according to memory usage, machine load and other factors and may be adjusted by altering environmental variables. In terms of memory usage, most variables in OpenMP code are visible to all threads by default. However, OpenMP provides a variety of options for data management, such as a thread-private memory and private variables, as well as multiple ways of passing values between sequential and parallel regions. Additionally, recent OpenMP implementations introduced the concept of tasks, as a solution for parallelizing applications that produce dynamic workloads. Thus, OpenMP is enriched with a flexible model for irregular parallelism, providing parallel while loops and recursive data structures.

The main advantage on using OpenMP is the ease of developing parallelismswith simple constructs that (often) do not differ too much from the original implementation.

The snippet at **Algorithm 3** shows the implementation used in this work: the matrix containing the distances between vertices is shared among all the threads, while the two nested for-loops are executed by each thread indipendentely. The k^{th} row is extracted by a single thread and shared among the other threads: this guarantees a sequential access to the k^{th} row faster than accessing the same row directly on M, resulting in a speedup of up to ~31%.

Despite the overhead that the dynamic scheduler entails, in this case it works better than a static one because the content of the sub-matrices varies from thread to thread so a group of threads may do more write operations on M and a static scheduler would make wait the others until they

Algorithm 3: Multithreaded FW with *OpenMP*

```
#pragma omp parallel num_threads(t)
 shared(M, K) private(k)
for $k = 1 \to n$ do
    #pragma omp master
    $K \leftarrow k^{th}Row(M, k)$
    #pragma omp for private(i,j)
     schedule(dynamic)
    for $i = 1 \to n$ do
        for $j = 1 \to n$ do
            if $M_{i,j} < M_{i,k} + K_j$ then
                $M_{i,j} \leftarrow M_{i,k} + K_j$
            end
        end
    end
end
```

finish. We benchmarked a loss in performance from ~7% to ~48% when choosing a static scheduler over a dynamic one, depending on the size of the matrix.
Also note that the solution does not implement a `collapse` directive because when we collapse multiple loops, OpenMP turns them into a single loop: there is a single index that is constructed from `i` and `j` using division and modulo operations. This can have huge impacts on the performance because of this overhead, especially if the matrix is really wide.

For the concrete implementation see **section 4.3**.

Figure 5 shows how 2 threads interact inside the matrix: the red and orange zones highlight the cells where Thread 1 and Thread 2 can write respectively; the blue and turquise cells represent the intermediate vertices that are compared with the vertice under analysis for Thread 1 and Thread 2 respectively.

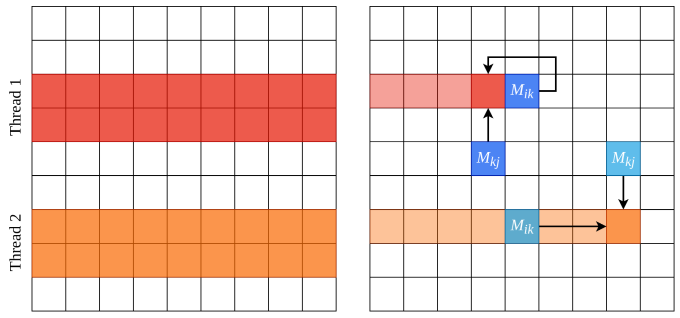

Figure 5: View of the data each thread can reach

We already proved that no race condition may appear in this case. No race condition can occur if k is the same among the threads and threrfore there's no need to verify atomicity of the write operation; the lack of OpenMP directive like `atomic` or `critical` plays in favor of performance.

The speedup of the solution is sligthy worse than the ideal speedup (see **Figure 6**)

When scaling from 7 to 8 threads, we notice a slight deviation from the previous (almost) linear trend. That's because the measurement is taken from a 8-core/8-thread CPU, namely Intel Core i7-9700K, and because no other cores were free to manage the OS and its subprocesses, the

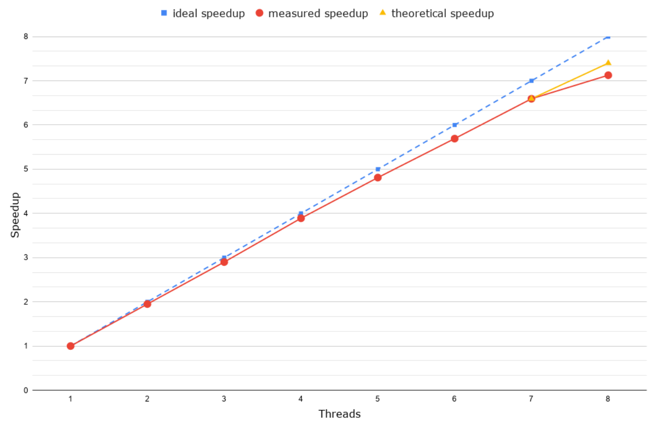

Figure 6: Speedup of *OpenMP* FW on a octacore CPU

scheduler divided this task among all the threads. So we interpolated the speedup, not counting the fluctuations due to the management of the OS.

The efficiency, which stays always above 90%, is shown in **Figure 7** alongside with its theoretical counterpart.

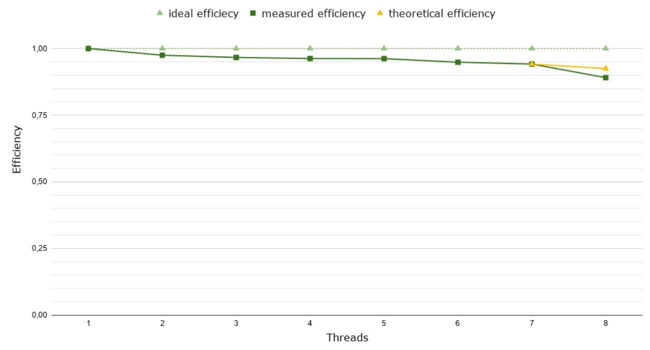

Figure 7: Efficiency of *OpenMP* FW on a octacore CPU

An overview of the timings collected can be found in **Table 4**.

3.4 GPGPU with CUDA

In recent years, many scientific and numeric GPGPU applications found success due to graphics hardware's streaming data-parallel organizational model.

The GPU serves, to an extent, as a coprocessor to the CPU programmed through the CUDA API. A single program known as kernel is compiled to operate on the GPU device to exploit the massive data parallelism inherit on Single Instruction, Multiple Data (SIMD) architecture. Groups of threads then execute the kernel on the GPU. Threads are organized into blocks which allow efficient sharing of data through a high-speed shared memory region accessible to the programmer directly through CUDA. Shared-memory is shared among threads in a block, facilitating higher bandwidth and overall performance gains. Therefore, algorithms must intelligently manage this fast shared memory cache effectively. This will fully utilize the data parallelism capabilities of graphics hardware and alleviates any memory latency that data intensive algorithms suffer from on the GPU.

Unlike the other two implementations, *CUDA* FW can rely on its *grid-of-blocks* system and thus the kernel can do no loops: k is still shared among threads so that no data race occurs but each thread instead of covering one or more rows, is assigned to a single cell.

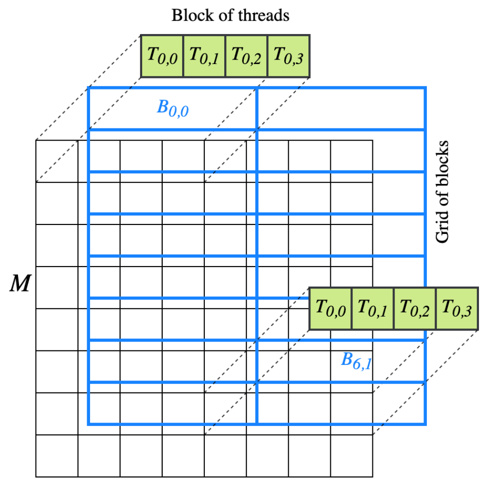

Figure 8: Mapping of CUDA threads inside M

Depending on the number of threads per block, each row of matrix M is covered by 1 or more blocks, that have always height equal to 1 thread. This mapping allows threads of the same block share some data that can be reused.
In this case the shared variable stores the value $M_{i,k}$ that is in common with the entire row. Because the shared memory offers lower latency and higher bandwidth than the global memory, it is worth storing this value for other threads.

Algorithm 4: Kernel of the *CUDA-FW* on a pre-Volta architecture

```
1  for $k = 0 \to n$ do
2      kernel<<<$\frac{n+b-1}{b} \times n, b$>>>($M, n, k$)
3  end
4
5  Function kernel($M, n, k$):
6      $j \leftarrow |B|_x \times B_x + T_x$
7      $i \leftarrow |B|_y$
8      if $j < n$ then
9          if $T_x = 0$ then
10             $K_{\text{shared}} \leftarrow M_{i,k}$;
11         end
12         syncthreads()
13         if $M_{i,j} < K_{\text{shared}} + M_{k,j}$ then
14             $M_{i,j} \leftarrow K_{\text{shared}} + M_{k,j}$
15         end
16     end
```

Algorithm 4 shows the kernel function for this implementation.
B_x is the coordinate x of the block, $|B|_x$ is the horizontal size of the block, T_x the coordinate x of the thread; $|B|_y$ is the coordinate y of the block and K_{shared} is the variable shared among all the threads of the same block.

One thread (namely $T_{0,0}$) reads from $M_{i,k}$ and stores the value in the shared memory. From here threads encounter a synchronization point which assures that $T_{0,0}$ had written in the shared memory and the block can actually use the correct value.

The state-of-the-art L1 cache in Volta and Turing offers lower latency, higher bandwidth, and higher capacity compared to the earlier architectures. Like Volta, Turing's L1 can cache write operations (write-through). The result is that for many applications Volta and Turing narrow the performance gap between explicitly managed shared memory and direct access to device memory [1].

For this reason we benchmarked **Algorithm 5** which relies only on global memory and we do not experienced any performance degradation on the Turing architecture. This mechanism is present in the newer Ampere architecture as well.

For the concrete implementation see **section 4.4**.

Algorithm 5: Kernel of the *CUDA-FW* on a Volta and post-Volta architectures

```
1  Function kernel($M, n, k$):
2      $j \leftarrow |B|_x \times B_x + T_x$
3      $i \leftarrow |B|_y$
4      if $j < n$ then
5          if $M_{i,j} < M_{i,k} + M_{k,j}$ then
6              $M_{i,j} \leftarrow M_{i,k} + M_{k,j}$
7          end
8      end
```

Because each thread does little computation and uses high memory bandwidth, timings are really low if compared to *serial-FW* and *OMP* FW with 8 cores.
In fact from **Figure 10** we see a speedup up to 87.8 times compared to *serial* FW and 12.34 times compared to *OMP* FW with 8 cores. We also notice that the maximum speedup is reached when the number of threads per block is equal to 128; that's because the occupancy of the Streaming Multiprocessors is at its peak and there are less wasted threads.

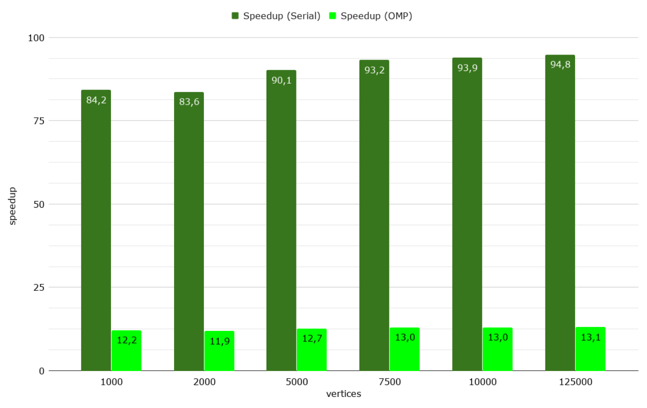

Figure 9: Speedup of *CUDA* FW (128 threads per block) consuming a medium matrix compared to *sequential* and *OMP* FW

Because the CUDA APIs don't allow the user to control the number of Streaming Multiprocessors or how many CUDA cores can be involved in the computation, the speedup is calculated based on the number of threads per block.

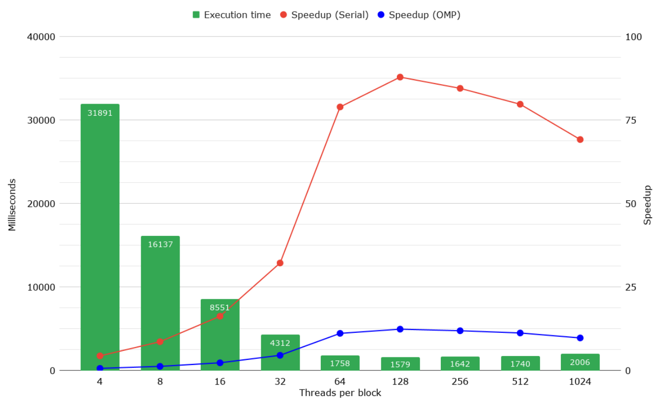

Figure 10: Efficiency of *MPI* FW consuming a medium matrix

3.5 Hybrid: MPI + OpenMP

In order to make the most of *MPI FW*, it can be combined with *OpenMP*; this makes sense because otherwise each process would run the i and j for loops on one thread and nowadays is more than rare to have a cluster composed of single thread CPUs.

Algorithm 6 shows how easily *MPI FW* can be modified so that each process can benefit from multithreading.

Algorithm 6: *MPI+OpenMP* FW

```
1  broadcast($n$, $ROOT$)
2  scatter($M$, $processes$)
3  #pragma omp parallel num_threads(t)
     private(k) shared(M, K, p)
4  for $k = 1 \rightarrow n$ do
5      #pragma omp master
6      if $rank$ = findOwnerOfKthRow($k, rank$) then
7          $K \leftarrow$ getRow($M, k$)
8      end
9      broadcast($K$)
10     #pragma omp for private(i,j)
         schedule(dynamic)
11     for $i = 1 \rightarrow \frac{n}{p}$ do
12         for $j = 1 \rightarrow n$ do
13             if $M_{i,j} < M_{i,k} + K_j$ then
14                 $M_{i,j} \leftarrow M_{i,k} + K_j$
15             end
16         end
17     end
18 end
19 gather($M$, $processes$);
```

For the concrete implementation see **section 4.5**.

3.6 Hybrid: MPI + CUDA

Another way to improve the efficiency of the cluster is to use GPGPU on each node.

Algorithm 7 shows how this solution can be implemented.

Although the increase of performance, this solution implies high costs because each node must be equipped with a GPU; moreover the GPUs (that must be produced by NVIDIA)

Algorithm 7: *MPI+CUDA* FW

```
broadcast($n$, $ROOT$)
scatter($M$, $p$)
for $k = 1 \rightarrow n$ do
    if $rank$ = findOwnerOfKthRow$(k, rank)$ then
        $K \leftarrow$ getRow($M$, $k$)
    end
    broadcast($K$)
    kernel<<<$\frac{n+b-1}{b \cdot p} \times n, b$>>>($M$, $n$, $k$)
end
gather($M$, $p$);

Function kernel($M$, $n$, $k$):
    $j \leftarrow |B|_x \times B_x + T_x$
    $i \leftarrow |B|_y$
    if $j < n$ then
        if $M_{i,j} < M_{i,k} + M_{k,j}$ then
            $M_{i,j} \leftarrow M_{i,k} + M_{k,j}$
        end
    end
```

must be compatibile with the version of the CUDA Toolkit for which the program was compiled for.

4 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 used for the analysis.

The results of each run are tested with 16bit Fletcher's checksum so that errors can be quickly checked over large matrices [2].

4.1 Serial implementation

The *sequential* version of the FW algorithm can be found here. The program compiles as 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 as 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 2 shows the execution time (expressed in milliseconds) depending on the number of vertices.

Figure 11 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:

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

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

Table 2: Execution time of the *serial* FW

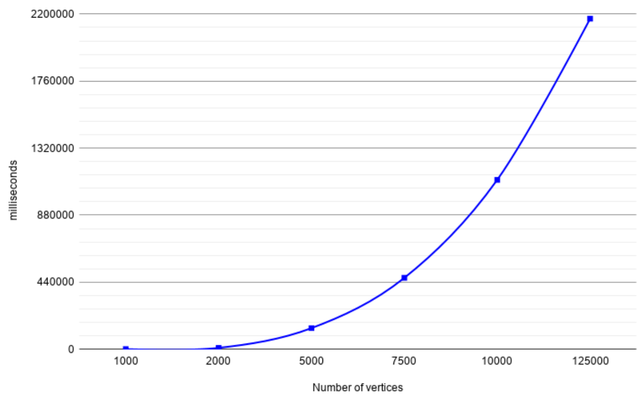

Figure 11: Trend of the execution time based on Table 2

4.2 MPI implementation

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

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

Like the *serial* version, the program accepts 2 arguments + 1 for `mpirun`:

```
$ 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 3 shows the execution time (expressed in milliseconds) and the percentage of time spent in initialization and communication, depending on the number of processors; in this case the program computed the *APSP* problem for 5040 vertices.

Processors	Execution time	MPI %
1	145447 ms	0.06
2	75059 ms	2.57
3	52336 ms	4.45
4	39962 ms	5.47
5	33184 ms	6.95
6	28731 ms	8.35
7	25853 ms	9.50
8	23120 ms	11.00

Table 3: Execution time of the *MPI* FW

Timings are captured through `mpiP` that calculates the percentage of time spent by MPI for initialization/finalization and communication. `mpiP` is a lightweight profiling library for MPI applications. Because it only collects statistical information about MPI functions, `mpiP` generates considerably less overhead and much less data than a tracing tools. All the information captured by `mpiP` is task-local. It only uses communication during report generation, typically at the end of the experiment, to merge results from all of the tasks into one output file.

4.3 OpenMP implementation

The *OpenMP* version of the FW algorithm can be found here. The program compiles as it follows:

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

Unlike the *serial* 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 4** shows the execution time (expressed in milliseconds) depending on the number of vertices and available cores.

Vertices	2 threads	4 threads	8 threads
1000	541 ms	364 ms	158 ms
2000	4365 ms	2921 ms	1260 ms
5000	69590 ms	46443 ms	19496 ms
7500	230260 ms	155531 ms	65118 ms
10000	543900 ms	367434 ms	153682 ms
12500	1063129 ms	716363 ms	299006 ms

Table 4: Execution time of the *OpenMP* FW

4.4 CUDA implementation

The *CUDA* version of the FW algorithm can be found here (pre-Volta architecture). Alternatively the version that uses only global memory is here (Volta and post-Volta architectures). The program compiles as follows:

```
$ nvcc cuda.cu -o cuda.out \
  -gencode=arch=compute_75,code=compute_75 -O3
```

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

```
$ ./cuda.out <v> <d> <b>
```

where `b` is the number of threads per block.
The `nvcc` version used for this work is 10.2 and the program ran on a CPU Intel Core i7-9700K and a GPU NVIDIA GeForce RTX 2070 Super. Table 5 shows the execution time (expressed in milliseconds) depending on the number of vertices and the block size of threads.

verteces	1024 block	256 block	32 block
1000	24 ms	19 ms	53 ms
2000	219 ms	186 ms	356 ms
5000	2818 ms	2709 ms	4966 ms
7500	9894 ms	9044 ms	16297 ms
10000	22190 ms	21392 ms	38208 ms
125000	42955 ms	41460 ms	75034 ms

Table 5: Execution time of the *CUDA* FW

Using a CPU-based timer would measure only the kernel's launch time and not the kernel's execution time and it would require a host-device synchronization point, like `cudaDeviceSynchronize()` which stalls the GPU pipeline.

So timings are taken with CUDA events that are created and destroyed with `cudaEventCreate()` and `cudaEventDestroy()`. `cudaEventRecord()` places the start and stop events into the default stream and the device will record a time stamp for the event when it reaches that event in the stream. The `cudaEventElapsedTime()` function returns in the first argument the number of milliseconds time elapsed between the recording events. This value has a resolution of approximately one half microsecond.

4.5 Hybrid implementation

The *MPI + OpenMP* version can be found here. The program compiles as follows:

```
$ mpicc -g -fopenmp hybrid.c -o hybrid.out -O3
```

The program accepts 4 arguments (see *MPI* and *openMP* versions):

```
$ mpirun -np <p> hybrid.out <n> <d> <t>
```

Conclusion

In this work we described three approaches to parallelize the FW algorithm with three different architectures: distributed with MPI, shared-memory multiprocessing with OpenMP and GPGPU with CUDA.

The fastest "pure" implementation is *CUDA FW* thanks to the high computation capability and high memory bandwidth, but it requires more expensive hardware.

MPI FW (with one thread per node) is still faster than the *serial*, but the implementation is more complex, the cost of the cluster (hosting and maintenance) makes the solution non convenient, the efficiency depends on the network bandwidth and the speedup is not that high.

OpenMP FW is almost 95 times faster than *serial FW* but 13 times slower than *CUDA FW*. The absence of overhead due to communication, fast memory access, easy development process and relatively low costs make this solution the most affordable, maintenable and cost-efficient one.

Talking about hybrid solutions, *MPI + CUDA* is obviously faster than *MPI + OpenMP* as long as the matrix is not small, but the benefits may not justify the total cost of the infrastructure: the monthly cost for hosting a server with a GPU is at least four times the cost of one without a GPU. This solution is suggested to those systems that really need the lowest response time possible (*e.g.* real-time systems or systems that cannot rely on a caching mechanism in front of them).

References

[1] NVIDIA Developer: Turing tuning guide,
`https://docs.nvidia.com/cuda/turing-tuning-guide/index.html`

[2] Fletcher, J. G. (January 1982). *An Arithmetic Checksum for Serial Transmissions.* IEEE Transactions on Communications. COM-30 (1): 247–252. doi:10.1109/tcom.1982.1095369