Parallelization Techniques for All-Pairs Shortest Path Algorithms

Yu-Po Wang
Department of Computer Science, National Yang Ming Chiao Tung University
Hsinchu, Taiwan
ypwang.cs09@nycu.edu.tw

Yu-Hung Kao
Department of Computer Science, National Yang Ming Chiao Tung University
Hsinchu, Taiwan
rb910518.cs09@nycu.edu.tw

Hsiang-Cheng Hsieh
Department of Computer Science, National Yang Ming Chiao Tung University
Hsinchu, Taiwan
hsianchengfun.cs09@nycu.edu.tw

ABSTRACT

The Floyd-Warshall (FW) algorithm is one of the most well-known algorithms that solve the all-pairs shortest path (APSP) problem. In this project, we adopt different approaches to parallelize the FW algorithm in three parallel programming environments: MPI, OpenMP, and CUDA. We conducted experiments using various data sizes to compare the performance differences between the serial FW algorithm and parallel versions. Additionally, we investigated and analyzed the performance variations in different parallelized FW algorithms or parameter settings. Consequently, we were able to fully utilize the underlying computing systems and achieve a significant performance upgrade for the FW algorithm.

ACM Reference Format:
Yu-Po Wang, Yu-Hung Kao, and Hsiang-Cheng Hsieh. 2024. Parallelization Techniques for All-Pairs Shortest Path Algorithms. In . HsinChu , Taiwan, 5 pages.

1 INTRODUCTION

The computational ability of processors has become increasingly powerful over the past few decades. However, the improvement of single-core processors has reached its physical limits of performance because increasing the clock frequency requires massive power consumption. To address this and to continue improving performance, the integration of multiple processing cores within a single processor has become a trend. On the other hand, different parallel programming environments, such as Message Passing Interface (MPI) for distributed memory systems, OpenMP for state-of-the-art multicore CPUs and CUDA for modern GPUs, have been developed. These programming models allow for the simultaneous execution of multiple tasks on different types of parallel architectures, thereby improving overall performance.

The All-Pairs Shortest Path (APSP) problem is a classic problem in graph theory that seeks to find the shortest paths between all pairs of vertices in a directed weighted graph. There is a well-known algorithm for solving this problem: the Floyd-Warshall (FW) algorithm. The input and the output of the FW algorithm are typically in matrix form, denoted as M. In this matrix, the entry in i^{th} row and the j^{th} column, denoted as M_{ij}, represents the distance from the node i to the node j in the graph. Despite its capability to solve the APSP problem, the FW algorithm is time-consuming. As shown in **Algorithm 1**, the FW algorithm has a time complexity of $\Theta(|V|^3)$ where V is the number of vertices in the graph. Consequently, the execution time of this algorithm escalates notably with the graph's size. In scenarios involving large graphs, its execution time may render it impractical for real-time applications. Hence, the inefficiency of the FW algorithm makes it a good choice for parallelization and optimization.

Algorithm 1 The Floyd-Warshall algorithm

```
for k = 1 → n do
    for i = 1 → n do
        for j = 1 → n do
            A_k[i, j] = min(A_{k-1}[i, k] + A_{k-1}[k, j], A_{k-1}[i, j])
        end for
    end for
end for
```

2 PROPOSED SOLUTIONS

In this project, we took two distinct approaches to parallelize the FW algorithm. The first approach involves naive parallelization of the inner loop, while the other is based on tiling. In this section, we introduce the rationale behind realizing the parallelization of the FW algorithm using these two approaches. Additionally, the implementation details in various programming environments will be presented in Section 4.

2.1 Parallize the inner loop

To parallelize an algorithm which is originally designed to run as a single-threaded program, determining the parallelizable regions is a critical consideration when developing its parallel version. There is a single large dependency in the FW algorithm, which is the outermost "k" loop. The loop cannot be done in parallel because each $k + 1^{th}$ iteration is dependent on the results generated by the preceding k^{th} iterations. However, the inner loop can be parallelized because each iteration independently updates the shortest path between pairs of nodes through a specific intermediate node. These updates are unrelated to previous iterations within the same inner loop, establishing their independence.

2.2 Tiling approach

After the preceding introduction to the FW algorithm, it can be found that there is a similarity it shares with the matrix multiplication—they both operate on matrices. Consequently, some optimization techniques useful for matrix operations can also be applied in the FW algorithm. One of the most common approaches is to split the matrix into several blocks and run the FW algorithm

independently on each block. Similar to matrix multiplication, tiling is more cache-friendly for CPUs by reducing cache misses and optimizing memory access patterns. In the case of GPUs, the shared memory in streaming multiprocessors (SM) is significantly faster than global memory. The tiled FW algorithm enables the movement of all data to the shared memory during the kernel execution in CUDA implementation, leading to improved overall performance due to faster memory access. However, the realization of the tiled FW requires some modifications that have to be made to the original algorithm. In the tiled FW, the blocks are not all independent, and the procedure is separated into three phases and repeated $\frac{n}{b}$ times, where n is the number of vertices in the graph, and b is the pre-defined block dimension. In the first phase, the k^{th} diagonal block is processed. In phase 2, the blocks in the k^{th} row and k^{th} column can be processed in parallel. Lastly, in phase 3, all remaining blocks can also be processed in parallel.

3 EXPERIMENTAL METHODOLOGY

1. **Environment:** We conducted our experiments on a PC with an Intel Core i5-12500H CPU @ 4.09GHz, 32GB RAM, and an NVIDIA GeForce RTX 3080 GPU. The operating system is Ubuntu 22.04.3 LTS. The algorithms are all implemented in C++. Furthermore, the experiments with MPI implementation were conducted on the workstation provided by the course.

2. **Test:** For the testing data, we generate random graphs using a fixed seed to ensure that the results are reproducible and comparable. Since the time complexity of the FW algorithm is irrelevant to the number of edges in the graph, we only generated test graphs with different numbers of nodes but maintain a consistent sparsity of 90%. Overall, our experiments are conducted with the following implementations: Serial FW implementation, Inner Loop-parallelized FW with OpenMP, MPI, CUDA, and Tiled FW with OpenMP and CUDA.

4 EXPERIMENTAL RESULTS

We first define the evaluation metrics for our experiments: speedup and efficiency. Speedup is the ratio of the baseline execution time to the algorithm execution time, while efficiency is the ratio of speedup to the number of processors. These two indices are used to evaluate the performance of the algorithms.

1. **Serial:** The baseline in our experiments is the serial FW algorithm shown in **Algorithm 1**. We also use the serial FW to generate the ground truth to ensure that our parallel implementations maintain the correctness. **Figure 1** shows the execution time required for serial FW on input graphs with different numbers of nodes, denoted as n. It is worth noting that the magnitude of the increase in execution time also grows as n increases, which implies that the serial FW would become extremely inefficient if n becomes too large.

2. **MPI:** MPI is a standardized message-passing system designed to function on a wide variety of parallel computing architectures. Therefore, the implementation of the parallelized FW for MPI in this project is executed on the workstation. We can see the result of the MPI implementation in the **Figure 2**. It is noteworthy that the efficiency of the MPI method with more than 4 processors is lower than 0.4, while the efficiency is relatively acceptable when the

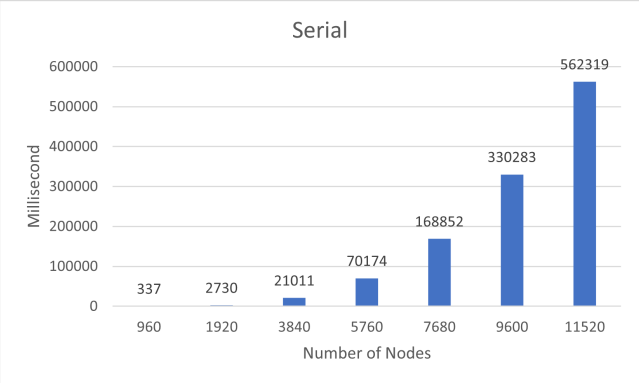

Figure 1: Serial

number of processors utilized for execution is less than 4. This may imply that when the CPU cores used are distributed across more than one machine, the efficiency would significantly deteriorate. We believe the underlying reason may be due to the communication overhead or the instability of the public server.

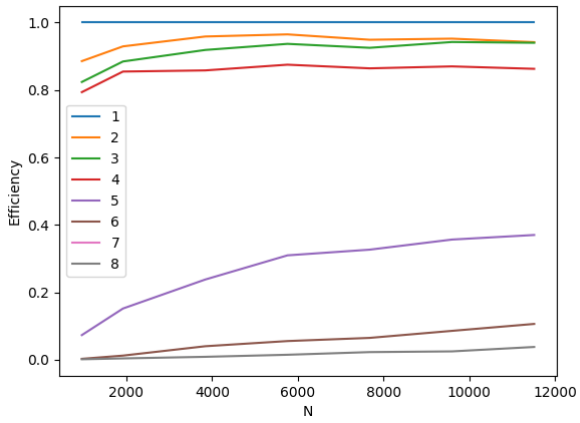

Figure 2: MPI efficiency

3. **OpenMP-Loop:** The snippet at **Algorithm 2** shows the inner loop-parallelized implementation using OpenMP in our project. As mentioned in section 2.1, the nested loop in the FW algorithm can be parallelized due to the independence of computation. Therefore, the realization is quite simple and straightforward because we only need to use the OpenMP directives to parallelize the i and j for-loops. The result is shown in **Figure 3**. It's evident that the speedup increases as the number of threads increases from 2 to 8, as measured on the 4-core/8-thread CPU in our PC.

4. **OpenMP-Tiling:** The snippet at **Algorithm 3** shows the implementation of the tiled FW using OpenMP. As blocks within the same phase can be processed simultaneously, each block is assigned to a thread in phases 2 and 3. **Figure 4** shows the performance comparison between the OMP-Tiling with different block size settings, suggesting that the required execution time decreases

Algorithm 2 Multithreaded FW with OpenMP

```
#pragma omp parallel num_threads(threads)
for k = 1 → n do
    #pragma omp for
    for i = 1 → n do
        for j = 1 → n do
            A_k[i, j] = min(A_{k-1}[i, k] + A_{k-1}[k, j], A_{k-1}[i, j])
        end for
    end for
end for
```

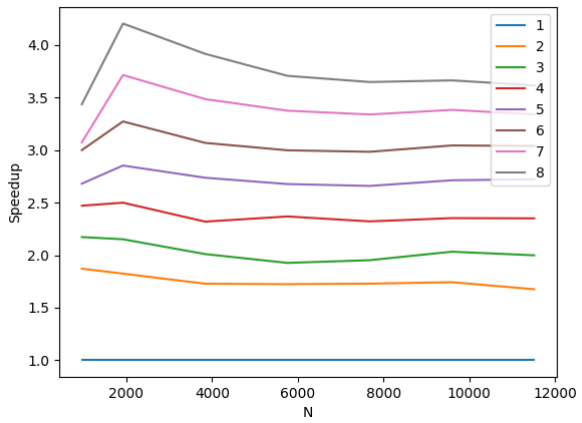

Figure 3: Speedup of OMP-Loop

as the block size increases. Furthermore, a comparison between OpenMP-Loop and Open-Tiling is shown in **Figure 5**. The result shows that Open-Tiling can perform better than Open-Loop in terms of execution time. This outcome aligns with our expectations, as the tiling technique can help to improve locality and reduce cache misses.

5. **CUDA-Loop:** The snippet at **Algorithm 4** shows the implementation of inner loop-parallelized FW using CUDA. In CUDA, a kernel is executed as a grid of thread blocks, with each block consisting of a predetermined number of threads. As a large number of lightweight threads can execute simultaneously, there are no loops in the kernel function. Instead, each thread is assigned to a matrix entry based on its block ID and thread ID. This differs from the CPU, where a thread usually covers one or more rows. **Figure 6** illustrates the comparison in performance between this CUDA implementation, the baseline serial implementation, and the OpenMP implementation. It is shown that CUDA-Loop can perform better than both the baseline and the OpenMP implementation and achieve over a 60x speedup and a 15x speedup respectively. This result meets our expectations since GPU is a highly data-parallel computer architecture and the independence of computation in the FW algorithm helps to fully utilize the hardware parallelism. However, there is a disadvantage to this parallelization approach with CUDA: all memory accesses are directed to global memory. According to the GPU's memory model, this is extreme slow and

Algorithm 3 Multithreaded Tiled FW with OpenMP

```
blocks ← n/b
for k = 1 → blocks do
    FW(M_kk)
    #pragma omp for
    for j = 1 → blocks do
        if j == k then
            continue
        end if
        FW(M_jk)
        FW(M_kj)
    end for
    #pragma omp for
    for i = 1 → blocks do
        for j = 1 → blocks do
            if i == k || j == k then
                continue
            end if
            FW(M_ij)
        end for
    end for
end for
```

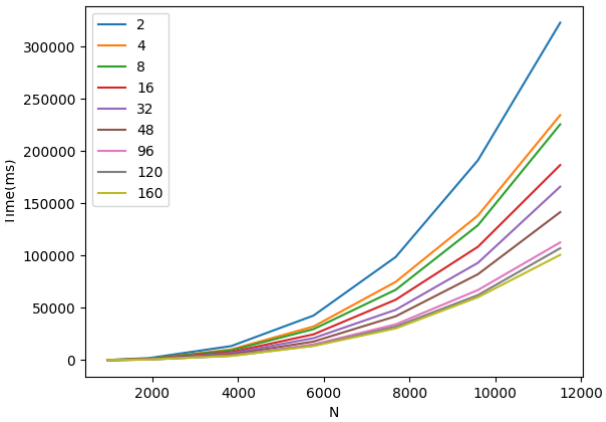

Figure 4: Execution time of OMP-Tiling

makes the FW algorithm memory-bound here. Additionally, it is noteworthy that CUDA-Loop reaches its peak performance when the number of thread per block equals to 256 instead of 512 or 1024. We found that the underlying reason is highly related to the value of theoretical occupancy, which is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. In our test, this value is at its highest when the number of threads equals 256, aligning with the result in **Figure 6**.

6. **CUDA-Tiling:** The snippets in **Algorithm 5** and **Algorithm 6** show the implementation of realizing tiled FW using CUDA. From **Algorithm 5**, we can see that there are three distinct kernel functions for the three phases. This difference arises because the

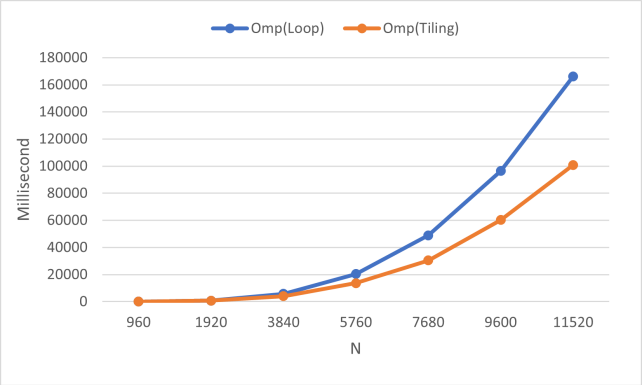

Figure 5: Comparison between OMP-Loop and OMP-Tiling

Algorithm 4 Kernel of CUDA-Loop

```
b ← ThreadsPerBlock
for k = 1 → n do
    KERNEL <<< (⌈(n+b−1)/b⌉, n), b >>> (M, n, k)
end for
function KERNEL(M,n,k)
    i ← |B_y| × B_y
    j ← |B_x| × B_x + T_x
    if j < n then
        if M_{i,j} < M_{i,k} + M_{k,j} then
            M_{i,j} ← M_{i,k} + M_{k,j}
        end if
    end if
end function
```

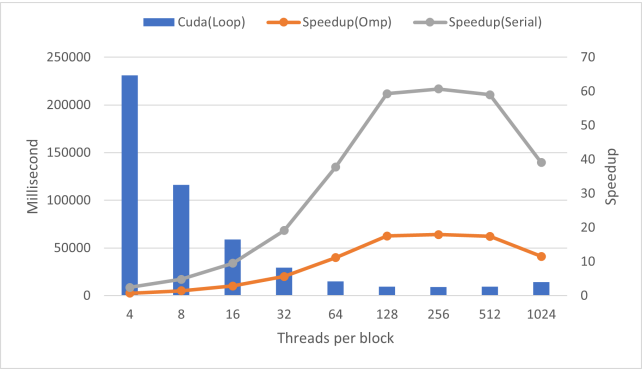

Figure 6: Cuda-Loop Speedup

number of blocks needing processing varies across the three phases in the tiling approach. Consequently, the three kernels must be implemented separately and in different fashions. Detailed implementations of these three kernels are provided in **Algorithm 6**. Despite their differences, all three kernels optimize memory access by leveraging the GPU's shared memory. This is achieved by each thread within the same block loading one element of each submatrix into the shared memory. Subsequently, each thread handles one position and operates for BLOCK_DIM iterations in the subroutine. Ultimately, all threads synchronize with each other and transfer the data in shared memory back to the global memory. Similar to **Figure 6**, **Figure 7** illustrates the performance comparison among this CUDA implementation, the baseline serial implementation, and the OpenMP implementation. It's noticeable that CUDA-Tiling also outperforms the baseline and the OpenMP implementation, achieving over 150x speedup and 50x speedup, respectively. This outcome also indicates that CUDA-Tiling surpasses CUDA-Loop, and it is consistent with our expectations, as shared memory access is significantly faster than global memory access.

Algorithm 5 Tiled FW with CUDA

```
blocks ← n/b
dimBlock ← (BLOCK_DIM, BLOCK_DIM)
dimGrid ← (blocks, blocks)
for k = 1 → blocks do
    KERNEL_PHASE1 <<< 1, dimBlock >>> (M, n, k)
    KERNEL_PHASE2 <<< blocks, dimBlock >>> (M, n, k)
    KERNEL_PHASE3 <<< dimGrid, dimBlock >>> (M, n, k)
end for
```

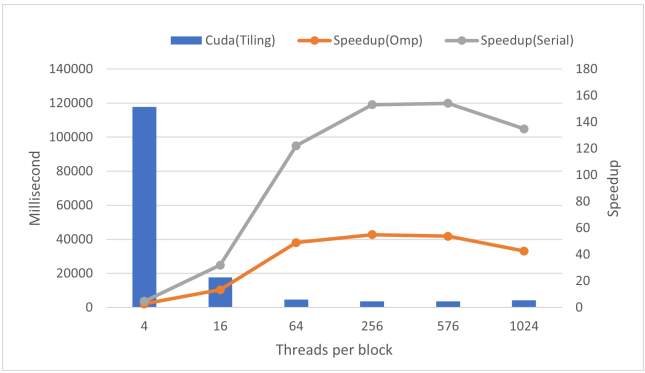

Figure 7: Cuda-Tiling Speedup

5 RELATED WORK

When researching parallelization techniques for all-pairs shortest path algorithms, it is important to review related work to understand existing approaches, and advancements in this field. Here are some of our potential related works:

1. **Parallel All-Pairs Shortest Path Algorithms:** Study classic algorithms like Floyd-Warshall and Johnson's algorithm and understand how they have been parallelized using CUDA, OpenMP, and MPI.

2. **CUDA-Based Parallelization:** Investigate research that focuses on parallelizing graph algorithms, including all-pairs shortest path, using CUDA. Explore the techniques and optimizations used in GPU-based implementations.[2]

3. **OpenMP Parallelization:** Look for research that discusses parallelization using OpenMP, especially in the context of graph algorithms and all-pairs shortest path. Examine load balancing and thread-level parallelism.

Algorithm 6 Kernel of the Tiled CUDA-FW

```
function FW_BLOCK_CALC(C,A,B,ty,tx)
    for k = 1 → BLOCK_DIM do
        A_idx ← ty × BLOCK_DIM + k
        B_idx ← k × BLOCK_DIM + tx
        sum ← A[A_idx] + B[B_idx]
        if C[ty × BLOCK_DIM + tx] > sum then
            C[ty × BLOCK_DIM + tx] = sum
        end if
        __syncthreads()
    end for
end function
function KERNEL_PHASE1(M,n,k)
    i ← k × BLOCK_DIM + T_y
    j ← k × BLOCK_DIM + T_x
    C_shared[T_y × BLOCK_DIM + T_x] ← M[i × n + j]
    __syncthreads()
    FW_BLOCK_CALC(C, C, C, T_y, T_x)
    M[i × n + j] ← C[T_y × BLOCK_DIM + T_x]
end function
function KERNEL_PHASE2(M,n,k)
    if B_x == k then
        return
    end if
    B ← BLOCK_DIM
    C_shared[T_y × B + T_x] ← M[B_x × B × n + k × B + T_y × n + T_x]
    B_shared[T_y × B + T_x] ← M[k × B × n + k × B + T_y × n + T_x]
    __syncthreads()
    FW_BLOCK_CALC(C, C, B, T_y, T_x)
    M[B_x × B × n + k × B + T_y × n + T_x] ← C_shared[T_y × B + T_x]
    __syncthreads()
    C_shared[T_y × B + T_x] ← M[k × B × n + B_x × B + T_y × n + T_x]
    A_shared[T_y × B + T_x] ← M[k × B × n + k × B + T_y × n + T_x]
    __syncthreads()
    FW_BLOCK_CALC(C, A, C, T_y, T_x)
    M[k × B × n + B_x × B + T_y × n + T_x] ← C_shared[T_y × B + T_x]
end function
function KERNEL_PHASE3(M,n,k)
    if B_y == k && B_x == k then
        return
    end if
    B ← BLOCK_DIM
    C_shared[T_y × B + T_x] ← M[B_y × B × n + B_x × B + T_y × n + T_x]
    B_shared[T_y × B + T_x] ← M[B_y × B × n + k × B + T_y × n + T_x]
    A_shared[T_y × B + T_x] ← M[k × B × n + B_x × B + T_y × n + T_x]
    __syncthreads()
    FW_BLOCK_CALC(C, A, B, T_y, T_x)
    M[B_y × B × n + B_x × B + T_y × n + T_x] ← C[T_y × B + T_x]
end function
```

4. **MPI Parallelization:** Explore how MPI has been used to parallelize graph algorithms, especially on distributed memory systems. Investigate strategies for data distribution and communication in MPI-based approaches.

5. **Hybrid Parallelization:** Consider work that combines multiple parallelization techniques, such as CUDA with OpenMP or MPI. Hybrid approaches can leverage both CPU and GPU resources effectively.

6.**Real-World Applications:** Seek examples of real-world applications that benefit from all-pairs shortest path algorithms. Investigate how parallelization techniques using CUDA, OpenMP, and MPI have been applied in these scenarios.

[1][3]

6 CONCLUSIONS

In this project, we implemented various parallelized versions of the Floyd-Warshall algorithm using different approaches and programming environments, including MPI, OpenMP and CUDA. Overall, the fastest implementation is CUDA-tiling, benefiting from the full utilization of GPU parallelism and the fast memory access provided by the shared memory, while CUDA-Loop has a inferior performance because all memory accesses are to the global memory. The parallelized FW with MPI is faster than the baseline serial FW. However, its efficiency deteriorates when the number of threads exceeds 4 because of the communication overhead or instability of the public workstation. For OpenMP, both OpenMP-Loop and OpenMP-Tiling can achieve significant speedups over the baseline, with Openmp-Tiling performing better than OpenMP-Loop due to its superior memory access pattern for caches. Additionally, the computation in FW algorithm has the potential for vectorization using SIMD instructions, but the implementation for CPU in this project does not utilize that capability. As a result, the future work for us in parallelizing the FW algorithm on CPU may involve leveraging SIMD instructions to achieve further speedup.

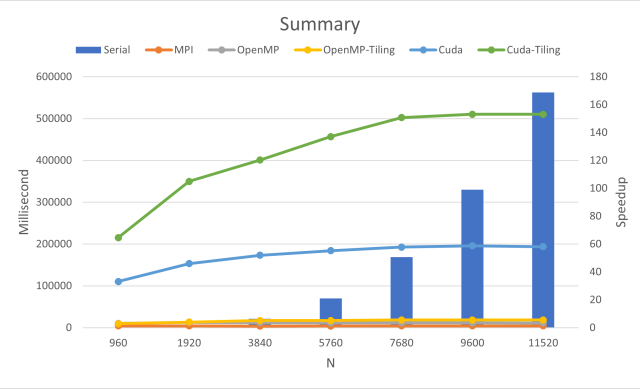

Figure 8: Summary

REFERENCES

[1] Asmita Gautam Dr. Russ Miller. [n. d.]. PARALLEL IMPLEMENTATION OF FLOYD-WARSHALL ALGORITHM. ([n. d.]).
[2] NVIDIA. [n. d.]. NVIDIA Developer: Turing tuning guide. ([n. d.]). https://docs.nvidia.com/cuda/turing-tuning-guide/index.html
[3] Students of the Parallel Processing Systems course School of Electrical Computer Engineering National Technical University of Athens. [n. d.]. Parallelizing the Floyd-Warshall Algorithm on Modern Multicore Platforms: Lessons Learned. ([n. d.]).