Parallelized Floyd-Warshall Algorithm: Implementation and Performance Analysis

Vinh Nguyen

April 2024

1, Introduction

The Floyd-Warshall algorithm is an important algorithm in graph theory and network analysis, providing a powerful method for determining the shortest paths between all pairs of vertices in a weighted graph.

The algorithm's sequential implementation, while efficient for small to moderate-sized graphs, can become computationally expensive for larger graphs. With the increasing prevalence of parallel computing in modern times, this project set out to find a way to parallelize the Floyd-Warshall algorithm.

This report presents a parallelized version of the Floyd-Warshall algorithm, using the OpenMP framework for parallelization. This project objective is to distribute the computational workload across multiple threads effectively, thereby reducing runtime and achieving improved scalability.

In this report, we provide a detailed overview of the parallelization strategy employed, implementation details, experimental setup, and performance analysis. We will also look at other parallelization strategies and provide a reason why we do not use them. Through experimental evaluation, we demonstrate the effectiveness of this parallelization in reducing runtime and improving the algorithm's scalability.

2, Floyd-Warshall Overview

The Floyd-Warshall algorithm is a dynamic programming-based method used to find the shortest paths between all pairs of vertices in a weighted graph. This property makes it useful for scenarios where the shortest path between any two vertices needs to be determined efficiently.

Algorithm 1 Floyd-Warshall Algorithm Pseudocode

```
for $k \leftarrow 0$ to $n - 1$ do
    for $i \leftarrow 0$ to $n - 1$ do
        for $j \leftarrow 0$ to $n - 1$ do
            if $graph[i][j] > graph[i][k] + graph[k][j]$ then
                graph[$i$][$j$] $\leftarrow$ graph[$i$][$k$] + graph[$k$][$j$]
            end
        end
    end
end
```

The algorithm operates by iteratively updating an n x n matrix, where n is the number of vertices in the graph. Each element (i, j) of the matrix represents the length of the shortest path from vertex i to vertex j. Initially, this matrix is filled with the weights of the edges in the graph. Then, through a series of relaxation steps, the algorithm iteratively improves the shortest path estimates until the optimal solution is obtained.

The key steps of the Floyd-Warshall algorithm can be summarized as follows:

Initialization: Initialize the distance matrix with the weights of the edges in the graph. If there is no direct edge between two vertices, the corresponding distance is set to 1,000,000,000.

Iterative Update: For each vertex k in the graph, consider all pairs of vertices (i, j) and update the distance matrix if a shorter path through k exists.

Termination: Repeat the iterative update step for all vertices k. After n iterations, the distance matrix contains the shortest path lengths between all pairs of vertices.

3, Parallelization Strategy

The Floyd-Warshall algorithm employs dynamic programming with a triple nested loop to compute all pairs' shortest paths within a graph. The outermost loop traverses all possible vertices, serving as the intermediate point in path calculations. Subsequently, the two inner loops iterate through all vertex pairs (i, j), updating the distances between them.

The inner two loops are inherently parallelizable as there are no inter-loop dependencies. Each iteration updates the distance between vertex pairs independently, making them suitable for concurrent computation. The graph can be subdivided into smaller segments, enabling parallel processing, provided each segment has access to the k-th row and column.

However, parallelizing the outermost loop presents challenges due to its sequential nature. Each iteration's calculation relies on the previous iteration's state, rendering straightforward parallelization impractical. Though alternative methods exist to partition the graph initially, those approaches are complex and time-consuming to implement. Extensive research and experimentation led us to choose a simpler but effective solution: parallelizing the inner two loops.

In this approach, the runtime is approximated as $O(N^3/p)$, where N represents the graph's vertex count and p denotes the number of threads working concurrently. While the outer loop remains sequential, accounting for O(N) time complexity, the parallelized inner loops, originally $O(N^2)$, benefit from parallel execution, resulting in $O(N^2/p)$ time complexity when evenly distributed among p threads.

4, Implementation Details

In this implementation, we use the OpenMP (Open Multi-Processing) framework for parallelization due to its simplicity and ease of integration with existing C/C++ codebases.

Algorithm 2 Parallel Floyd-Warshall Algorithm Pseudocode

```
#pragma omp parallel num_threads(numThread) shared(graph, n)
for $k \leftarrow 0$ to $n-1$ do
    #pragma omp for private(i, j) schedule(dynamic)
    for $i \leftarrow 0$ to $n-1$ do
        for $j \leftarrow 0$ to $n-1$ do
            if $graph[i][j] > graph[i][k] + graph[k][j]$ then
                graph$[i][j] \leftarrow$ graph$[i][k]$ + graph$[k][j]$
            end
        end
    end
end
```

We wrap the entire algorithm within an OpenMP parallel region, where the number of threads is determined by the numThread parameter provided by the user. Within this parallel region, both the graph data and the size of the graph (n) are shared among all threads to facilitate collaborative computation.

Parallel Region Initialization: We begin by entering an OpenMP parallel region, where some subsequent computations will be executed in parallel by multiple threads. The shared(graph, n) clause ensures that the graph data and the size of the graph (n) are accessible to all threads within the parallel region.

Sequential Execution over k: The outer loop iterates sequentially over each vertex k in the graph.

Inner Loop Parallelization over i: Within each iteration of the outer loop, the inner loop over i is parallelized using the #pragma omp for directive. Using private(i, j) and schedule(dynamic), the threads maintain their own copy of the loop iterators and achieve load balancing and efficient resource utilization. This parallelization strategy distributes the workload of updating the shortest paths among multiple threads. Each thread independently processes a subset of iterations of the inner loops, effectively parallelizing the computation of shortest paths for different pairs of vertices.

5. Experimental Setup

For our experimental setup, we developed a Python script to generate a diverse set of test cases. These test cases encompass different graph sizes, ranging from 10 to 2000 vertices, to thoroughly evaluate the performance of the parallelized Floyd-Warshall algorithm.

Each test case is formatted as follows: the first line indicates the number of vertices, denoted by N. Subsequently, there are N lines, each representing a row in the graph matrix. Each line contains N entries, where the j-th entry on line i-th denotes the weight of the edge from vertex i to vertex j. If there exists a path between vertex i and vertex j, the weight (`graph[i][j]`) is randomized from 1 to 10,000. If there is no direct path between vertex i and vertex j, the weight is set to a large value of 1,000,000,000 to denote infinity. Additionally, when i equals j (i.e., the diagonal entries), the weight is explicitly set to 0 to represent the absence of any edge from a vertex to itself.

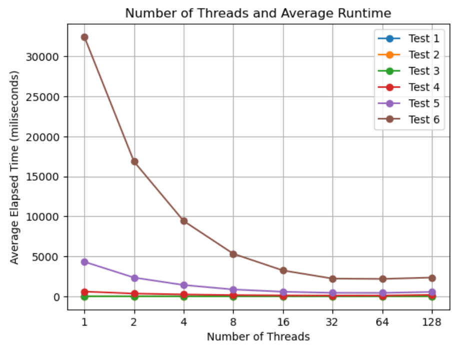
The rationale behind choosing 1,000,000,000 as the maximum weight is to ensure compatibility with the comparison operation in the shortest path calculation: `graph[i][j] > graph[i][k] + graph[k][j]`. The maximum possible sum here can reach up to 2,000,000,000, which remains within the integer limit.

Subsequent to test case generation, we created a script to execute the Floyd-Warshall algorithm across all test cases, varying the number of threads from 1 to 128. Each test case underwent 10 iterations per thread count. When the thread count was set to 1, the program executed the sequential version of the algorithm. The output of each execution was logged into 6 distinct output files, aligning with the 6 test cases.

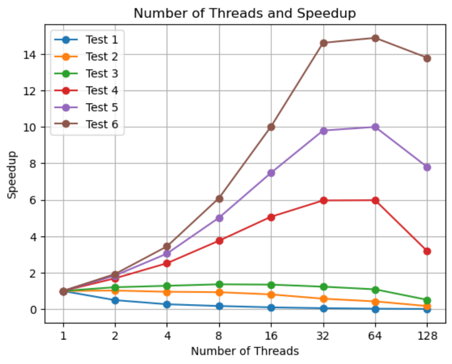
Following the execution phase, we implemented another script to reduce the output files into a comprehensive text file. We utilized the aggregated data to generate graphs illustrating the runtime and speedup achieved by each thread count across all test cases.

6. Experimental Results

After undergoing data transformations, the conclusion of our experiment is shown in graphical representations. These graphs filter complex information into visual insights, showing the trends and patterns.



(a) Average Runtime



(b) Speedup

Figure 1: Runtime and Speedup

7. Analysis and Discussion

Before beginning this section, we wish to clarify the number of vertices in each test case: test 1 has 10 vertices, test 2 has 50 vertices, test 3 has 100 vertices, test 4 has 500 vertices, test 5 has 1000 vertices, and test 6 has 2000 vertices. We have thoroughly tested the results of the parallelized version to ensure that the output graph is correct. Therefore, in this section, we will focus on the runtime and speedup of the algorithm.

From the average runtime figure (figure 1a), we observe that the average runtime of test 6 is significantly higher than that of the other five test cases. At 1 thread, test 6 takes more than 30,000 milliseconds to execute, while the other five test cases take less than 5,000 milliseconds. Even test case 5, which has 1,000 vertices—half the number of vertices in test 6—runs approximately 6 times faster. One explanation for this phenomenon is that the runtime grows at a rate proportional to the square of the number of vertices. A notable trend is that when the number of threads is increased to 2, the runtime is reduced to almost exactly half. However, as more threads are added, the trend plateaus, observed most clearly in test 6.

From the average speedup figure (figure 1b), we can see that the speedup trend becomes more pronounced in higher test cases. In test cases 1, 2, and 3, we observe almost no speedup as the number of threads increases. Once we reach test case 4 and above, the effect of using more threads becomes more evident. However, all three of these tests experience a plateau at 64 threads and take a slight dip at 128 threads. This dip is smaller for higher test cases. For smaller test cases, we suspect that because the graph size is too small, the overhead of allocating and managing threads can be comparable to the time taken for sequential execution. The slight dip observed from test cases 4 and above may be attributed to the diminishing returns of adding threads due to synchronization and communication overheads among threads, which becomes significant when the number of threads exceeds the hardware's capabilities to efficiently manage them.

8. Conclusion

This project has demonstrated the feasibility and effectiveness of parallelizing the Floyd-Warshall algorithm using OpenMP to enhance its performance for solving the all-pairs shortest paths problem in graphs. The implementation was optimized by focusing the parallelization efforts on the inner two loops of the algorithm while keeping the outer loop sequential to maintain the integrity of the algorithm's dependencies.

The experiments, conducted across a range of graph sizes from 10 to 2000 vertices, reveal that parallelization significantly reduces computation time, especially as the size of the graph increases. However, the observed plateau in performance improvement with increasing threads beyond a certain point highlights the limitations imposed by overhead costs associated with thread management and synchronization.

In conclusion, the parallelized Floyd-Warshall algorithm presents a solution for improving the performance of all-pairs shortest path computations on large graphs.

9, Future Work

Future work could explore more sophisticated partitioning strategies and dynamic scheduling techniques to further optimize the distribution of computational tasks among threads. Additionally, exploring the implementation on different hardware architectures and using other parallel computing frameworks might offer insights into achieving greater efficiencies.

10, References

[1] Wikipedia contributors. "Parallel All-Pairs Shortest Path Algorithm." *Wikipedia, The Free Encyclopedia.* Wikimedia Foundation. Web. `https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm`.

[2] Moore, J. S. (n.d.). "Parallel Algorithms for All-Pairs Shortest Paths". Retrieved from `https://moorejs.github.io/APSP-in-parallel/#Approach`.

[3] Students of the Parallel Processing Systems course, School of Electrical & Computer Engineering, National Technical University of Athens. (Year). "Parallelizing the Floyd-Warshall Algorithm on Modern Multicore Platforms: Lessons Learned".