Parallelizing All-Pairs Shortest Path

Elias Haddad[1], Stela Ciko[1]
[1]University of Rochester, Rochester, NY

ABSTRACT

This paper analyzes the Floyd-Warshall algorithm that computes the All-Pairs Shortest Path (APSP) problem for directed weighted graphs and utilizes techniques in parallel computing to parallelize the algorithm. We employ concepts such as 2D Block Mapping and Barriers to accomplish this with promising results. We also improve our previous benchmark test results by employing Thread Pooling techniques. We achieve a final asymptotic runtime from $O(n^3)$, where n is the number of vertices, to $O\left(\frac{n^3}{p}\right) + (p * T_{Creation})$ where p is the number of threads and $T_{Creation}$ is an overhead constant from initializing p number of threads.

Keywords: Parallelization, All-Pairs Shortest Path, Floyd Warshall, Pooling, Multi-Threading

I. INTRODUCTION

A. Floyd-Warshall and APSP

The All-Pairs Shortest Path (APSP) problem in directed weighted graphs has the goal of finding the shortest path between each vertex to every other vertex in the graph. The Floyd-Warshall (F-W) algorithm is a well-established method for solving this problem and traditionally exhibits a cubic time complexity of $O(n^3)$. It works by identifying an intermediate vertex (call it k) in the graph and checking every possible combination of vertices and taking the minimum weight of a direct connection or an indirect connection through k. Furthermore, the algorithm considers each vertex in the graph as an intermediate vertex and thus, k iterates through vertices 1 to n.

We identified this algorithm as a suitable candidate for parallelization due to the expensive computation of finding the combinations of all these vertices per value of k. Since this can be done in any order within each kth iteration, we can parallelize this portion of the algorithm without any synchronization or broadcasting necessary.

Figure 1 shows the serial implementation of the F-W algorithm, but with the outer and two inner loops separated to emphasize the loop-carried dependence in the outer loop, and independence in the two inner loops. The algorithm accepts a weighted graph and will simply transform it into a weighted matrix (W) that the main algorithm will use. For simplicity in our testing, we assume that -1 represents a value of infinity, signifying there's no edge between two vertices.

```
Algorithm 1 Floyd-Warshall
Require: $n \geq 1$ (number of vertices), weight matrix $W$ of dimension $n \times n$
Ensure: Matrix $D$ of shortest paths distances (output)
  $D \leftarrow W$
  for $k = 1$ to $n$ do
      ProcessData($k$)
  end for

  function ProcessData($k$)
      for $i = 1$ to $n$ do
          for $j = 1$ to $n$ do
              $D[i,j] \leftarrow \min(D[i,j], D[i,k] + D[k,j])$
          end for
      end for
  end function
```

Figure 1. F-W APSP algorithm split into its outer and inner loops to highlight dependencies.

B. Dependency Overview

We identified a series of dependencies that would be critical to consider in order to ensure the correctness of the algorithm is preserved. The first is the loop-carried dependence mentioned in the prior section. This is a rather intuitive dependence, especially since the outer loop must consider intermediate vertices in a specific order and when the inner loop tests candidates, the kth row and column must be up to date from previous iterations.

Other methods[1] consider broadcasting methods within the inner loop as means of synchronization between threads as they compute their portion of W. We claim that such means are not necessary to ensure the correctness of the algorithm and that no synchronization is required in the inner loops. This is due to our observation that at some value of k, the kth row and column are never written to but only read from. We can consider four possible cases to support our claim for some values of i and j in the inner loop:

Case 1: $k = i, k = j$

Both the direct and intermediate paths are self-looping so nothing gets updated/written.

Case 2: $k = i, k \neq j$

We'd consider the minimum weight of the path $i \rightarrow i \rightarrow j$ and $i \rightarrow j$ meaning we will always choose the direct path that's already set. No writes happen.

Case 3: $k \neq i, k = j$

We'd consider the minimum weight of the path $i \rightarrow j \rightarrow j$ and $i \rightarrow j$ meaning we will again always choose the direct path that's already set. No writes happen.

Case 4: $k \neq i, k \neq j$

This is obvious since whatever path we're potentially writing to isn't on the kth row or column.

With these dependencies considered, we proceed in the remaining sections to discuss our methods and results for our parallelization implementation.

II. METHODS

As mentioned in the prior sections, we employ a series of techniques to parallelize the F-W algorithm which we will discuss in detail here.

A. 2D Block Mapping

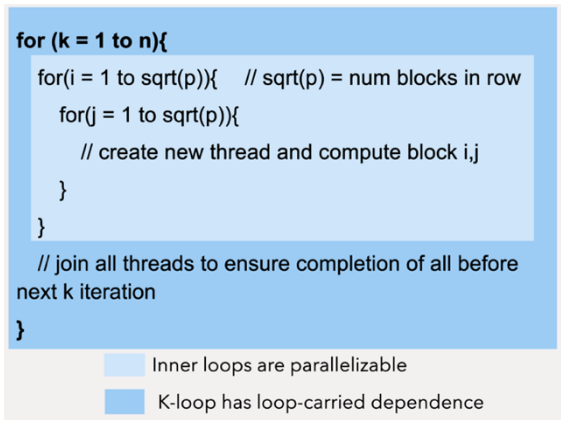
We employ this technique to partition W into "blocks" that have a 1:1 relationship with the number of threads. In particular, each thread is assigned a unique block to run the inner loop portion on for each k-loop iteration. The equation for partitioning the matrix is as follows:

$$b = \frac{n}{\sqrt{p}}$$

Where n is the size of an $n \times n$ weight matrix, and b is the size of a $b \times b$ sized block. To scale this partitioning to larger inputs, we can simply solve for p given that b is set to some feasible constant. This will allow the number of threads to scale to a maximum of n^2, or the number of cells in the matrix.[2]

B. Method 1 - Simple Parallelization

Our first parallelization attempt is a simple approach that constructs p threads to compute each block of the matrix. We construct the threads inside the j loop, allowing each block i,j to be computed by a thread. To ensure that all threads are done before we move to the next k iteration, we join all threads once the i and j loops are done, acting as a barrier for the threads. Pseudocode for this method is shown in Figure 2.



```
for (k = 1 to n){
    for(i = 1 to sqrt(p)){    // sqrt(p) = num blocks in row
        for(j = 1 to sqrt(p)){
            // create new thread and compute block i,j
        }
    }
    // join all threads to ensure completion of all before
next k iteration
}
```

Inner loops are parallelizable
K-loop has loop-carried dependence

Figure 2. Simple parallelization: create and destroy p threads at each iteration of k.

C. Barriers

Utilizing barriers in our parallelization methods is necessary to ensure correctness. This is because threads that finished computation at some iteration of k can progress onto the next k iteration. If this happens, they may potentially read old values due to other threads not being done with the previous iteration.

The solution to this in the first parallelization method was to join all threads at the end of each k loop. However, this method repetitively creates and destroys p threads for each iteration of k, resulting in an unnecessary overhead.

In the second parallelization method which we describe below, we create a barrier using condition variables, locks, and mutexes. We also introduce an atomic variable to keep count of the number of tasks completed. This value is incremented once a thread is done computing its block. The barrier at the end of each k loop blocks computation until it is notified that the number of tasks completed equals p, the number of running threads. This signifies that all threads are done and allows the next k iteration to take place. This barrier is used alongside the method described below.

D. Method 2 – Thread Pooling

To prevent the overhead from creating and destroying threads at each k iteration, we instead create a pool of p

threads and reuse them through iterations. We also maintain a queue of tasks that need to be computed. Free threads will pick up a task and execute it. As seen in Figure 3, instead of directly spawning a thread to compute a block as in the previous attempt, we add the task of computing this block to the queue of tasks so that it is computed by a free thread. At the end of the k loop, we block in the barrier described above until all threads are done computing. This algorithm reuses threads and doesn't destroy them until all iterations of k are finished. Once it finishes, all threads are joined.

```
// create pool of p-threads
// create barrier
for (k = 1 to n){
  for(i = 1 to sqrt(p)){     // sqrt(p) = num blocks in row
    for(j = 1 to sqrt(p)){
      // enqueue task of computing block i,j
    }
  }
  // block in barrier until notified that all p-threads
    finished computing
}
```

Figure 3. Parallelization with a thread pool: enqueue task to compute block i,j.

III. RESULTS AND DISCUSSION

This section will detail our benchmark results from the serial implementation of the algorithm, simple parallelization method, and the optimized thread-pooling method for graphs of size n=100 to n=1414. For each graph, we tested on various number of threads. Since the number of threads is the number of blocks, the number of threads we tried are the divisors of the graph size for an even block split. We gathered the average serial runtime for each graph, and the average parallel and parallel pool runtimes for each number of threads. The table and graphs below use only the parallel runtimes as found by the number of threads that result into the shortest runtime. Figure 4 reports the results for each graph, as well as the speedup for both parallel methods. Both methods display a speedup, with the second method giving more promising results.

	Runtime (ms)				
# Vertices	Avg Serial	Smallest Parallel (no pool)	Smallest Parallel Pool	Serial/No Pool Speedup	Serial/Pool Speedup
100	5.623	9.692	5.064	0.58	1.11
318	117.816	52.435	41.59	2.25	2.83
512	471.694	162.163	144.27	2.91	3.27
729	1332.827	489.127	487.616	2.72	2.73
1000	2436.343	609.683	449.2	4.00	5.42
1414	5885.02	1691.843	1509.62	3.48	3.90

Figure 4 Runtime results for each graph.

A. Serial vs. Simple Parallel vs. Parallel with Thread Pooling

Comparing the three methods, we found that the parallel solutions outperform the serial significantly as the number of vertices increases. The parallel attempt with the pooled threads is also faster than the simple parallelization as graph size increases. This is shown in Figure 5.

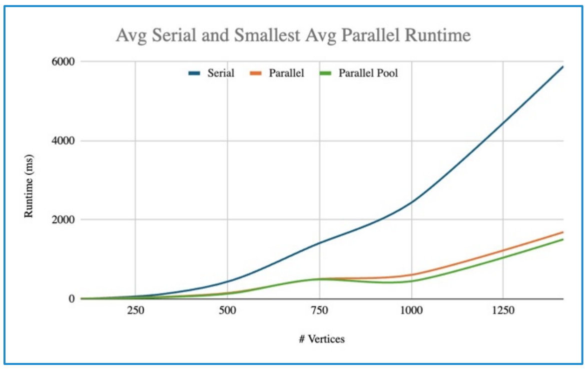


Figure 5. Serial vs. Parallel Runtimes.

B. Serial vs. Parallel with different number of threads

The serial version was faster than both parallel solutions for N=100. The benefits of the parallel solutions kick in with greater graph sizes. As shown in Figure 6, the parallel solutions are significantly faster than serial, and parallel pool is better than the simple parallelization. Relatively smaller number of threads/blocks tend to be better.

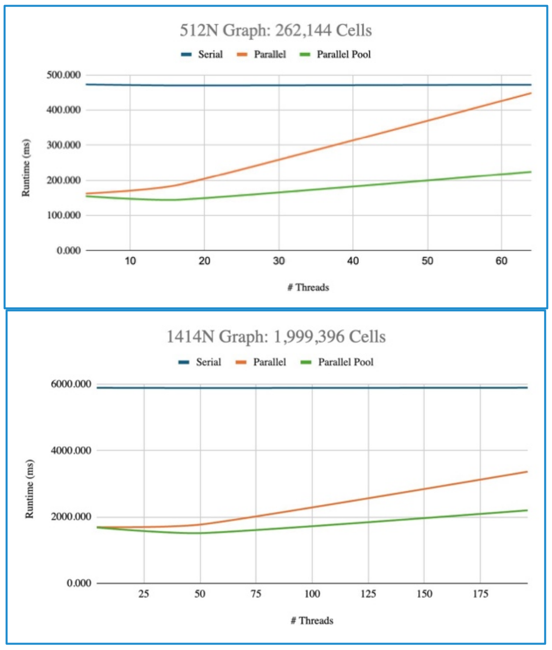


Figure 6. Results for N=512 and N=1414 Graphs.

IV. CONCLUSION

Based on our proposed parallelization method for the F-W algorithm and our test results, our implementation achieved promising speedups for computing the APSP for directed weighted graphs. We observed speedups as large as 5.42 times for graphs of size *n=1000* from our enhanced parallelization method utilizing thread pooling.

Our two parallelization methods both performed well, especially with large inputs, with the thread pooling enhanced method slightly outperforming the traditional implementation on average due to a decrease in overhead from thread creation. We observe the difference in runtime between the two methods steadily increasing with the number of threads used and the size of the graph. This is consistent with our enhanced runtime of
$O\left(\frac{n^3}{p}\right) + (p * T_{Creation})$, compared to a slower runtime of
$O\left(\frac{n^3}{p}\right) + n(p * T_{Creation})$ for the first method.

We anticipate conducting future analysis into the performance of our implementations on sparse and dense graphs, which we expect to see speedups for both but less for the former. Furthermore, we will also consider utilizing GPU parallelization and distributed system capabilities in hopes of observing larger speedups.

V. REFERENCES

[1] Gautam, Asmita. Parallel Implementation of the Floyd-
Warshall Algorithm, 2019, cse.buffalo.edu/faculty/miller/Courses/CSE633/Asmita-Gautam-Spring-2019.pdf.

[2] "Parallel All-Pairs Shortest Path Algorithm." Wikipedia, Wikimedia Foundation, 23 Apr. 2024, en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm#:~:text=Another%20variation%20of%20the%20problem,single-source%20shortest%20path%20algorithm.