HPC Project

Floyd-Warshall Algorithm

Dharam Soni (20190802088)

Suman Kumar (20190802080)

Serial Algorithm

The Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). This algorithm will find the shortest distance possible to any of the two nodes. Our aim is to optimize the cache usage in such a way that we achieve a speedup as close as possible to Amdahl's upper bound law.

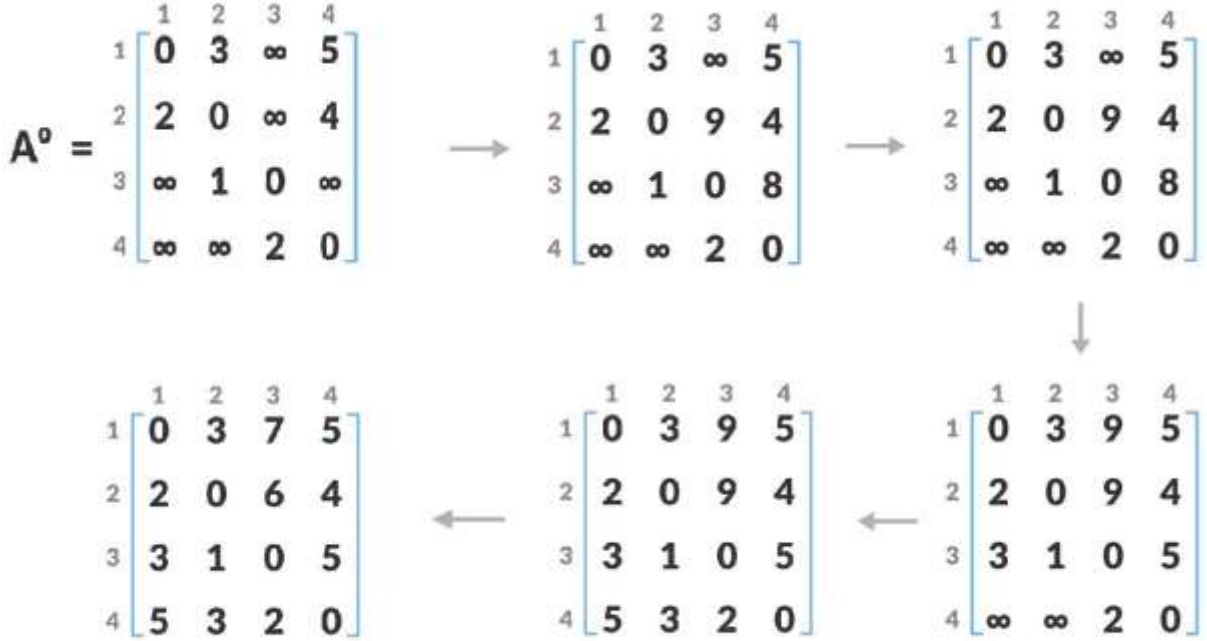
Input

The input contains a graph of N nodes and their edge weights in the form of an adjacency matrix of size N x N. This algorithm can handle negative edge weights but not negative cycles since if there does exist a negative cycle then we can circle to this cycle, go around it an infinite number of times and have a distance of $-\infty$ between all nodes.

Output

The output is a matrix dis of size N x N. This means that the optimum distance for going from node i to j is the value dis[i][j]. Again in case of an undirected graph, we will get dis[i][j] = dis[j][i].

Example - Serial Execution



Source : https://www.programiz.com/dsa/floyd-warshall-algorithm

Algorithm Complexity

```
for(k=0; k<N; k++){
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            if(dis[i][j] > dis[i][k]+dis[k][j]){
                dis[i][j] = dis[i][k]+dis[k][j];
            }
        }
    }
}
```

Floyd Warshall Algorithm consists of three loops over all the nodes.

The inner most loop consists of only constant complexity operations.

Hence, the asymptotic complexity of Floyd Warshall algorithm is $O(n^3)$.

Here, n is the number of nodes in the given graph.

Thus for the larger input size, serial code takes a lot of time to compute

Parallelization Strategy

We found out that the algorithm spends considerable time in calculating the offsets for dis[i][j] by doing i*N+j.

We can remove those computations by storing the required auxiliary arrays and then we can access those auxiliary arrays using the indexes i and j only. This makes sure that the approach remains parallelizable.

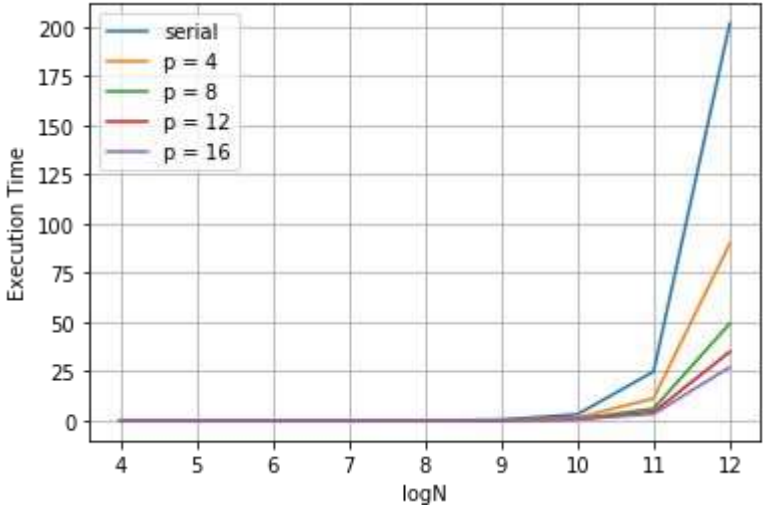
```
for(k=0; k<N; k++){
    #pragma omp parallel for private(i, j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            dis[i][j] = min(dis[i][j],dis[i][k]+dis[k][j]);
        }
    }
}
```

Approach 1

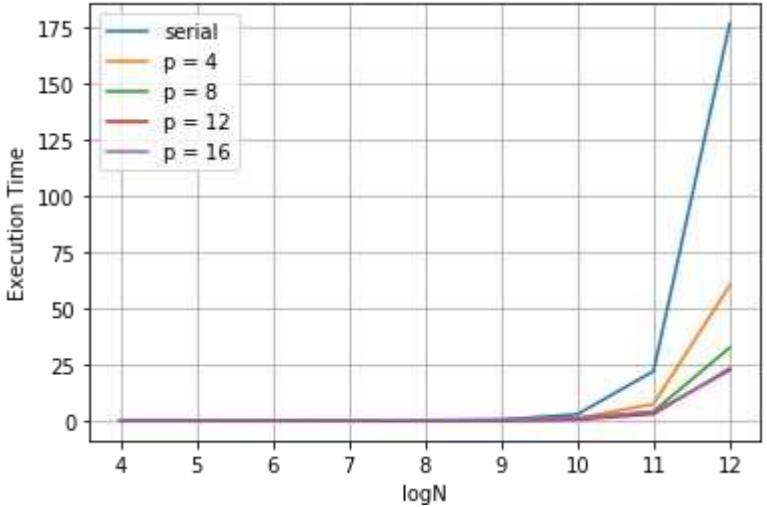
```
for(k=0; k<N; k++){

#pragma omp parallel for schedule(static,chunk) private(i)
    for(i=0; i<N; i++){
        x[i] = dis[k][i];
        y[i] = dis[i][k];
    }

#pragma omp parallel for schedule(static,chunk) private(i, j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            if(dis[i][j] > y[i]+x[j]){
                dis[i][j] = y[i]+x[j];
            }
        }
    }

}
```

Approach 2

Execution Time

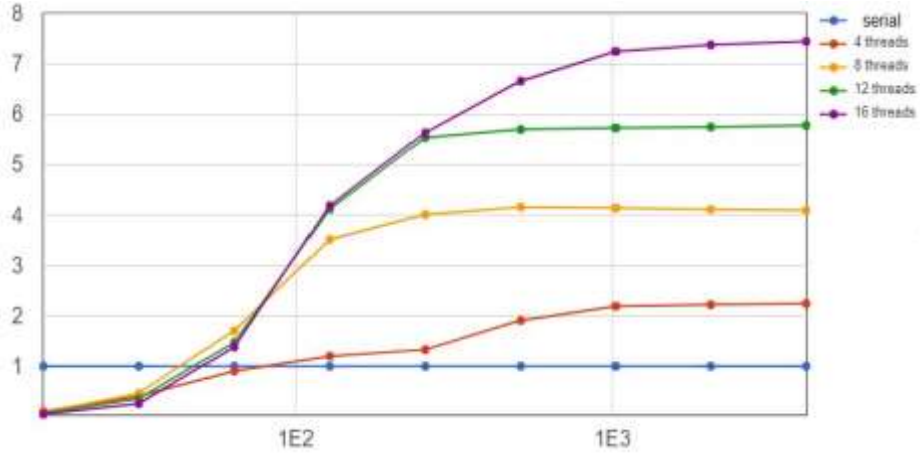


Naive Parallelization) ~201 s

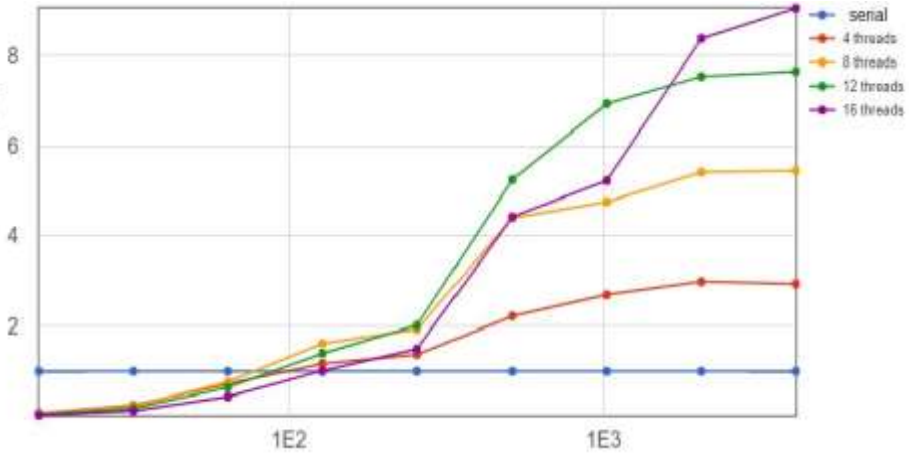


Using Auxiliary Arrays) ~176 s

Speedup



Naive Parallelization) ~2.24



Using Auxiliary Arrays) ~2.94

Analysis

- Cache Misses
  - Naive Approach: For each iteration of k we get misses when dis[i][k] (column k) and dis[k][i] (row k) are accessed. Once they are accessed, they are available in cache. But for the first time when threads begin to access row k, multiple threads might request the same data resulting pN/16 misses at the worst case instead of N/16 misses in the serial approach. This results in a lower speedup.
  - Using Auxiliary Arrays: The problem that we faced in the naive approach doesn't appear here since the auxiliary arrays are loaded in the cache before any thread begins to compute the updates. This means that there is no collision of data fetches and hence we save on cache misses
- Compute Costs
  - Naive Approach: As discussed earlier, we incur $2N^3$ multiplication instances when we access dis[i][k]
  - Using Auxiliary Arrays: These computations are reduced by $2N^2$ computations when we use the auxiliary arrays

Analysis - Cont.

- Scheduling Policies (Dynamic vs Static)
  - Since dynamic scheduling divide the workload among thread according to their availability, every time a thread is idle it has to be assigned to a chunk by the OS. This scheduling will cause a delay in the run time called as "Scheduling Overhead".
  - Load balance means "Equal work to all threads". Dynamic scheduling automatically assigns the work to that thread which is available causing no delay in the run time. Whereas static scheduling algorithm assign the work depending upon the chunk size.

Since we observed that there are no condition checks inside the parallel region and all thread have to do a fix amount of work on each iteration. Thus the workload can be evenly distributed among threads at compile time and we can get rid of dynamic scheduling overhead.

Conclusion

Speedup: We see that we fail to obtain the theoretical speedup of p. This is because of the parallel overhead which also includes the synchronisation step that occurs at each iteration. It is possible that threads might have to wait to achieve the synchronisation. This introduces wait time which can't be removed. It is also possible that due to multiple users on the cluster, threads don't get assigned the same core after context switching and miss out on the cache hits that were anticipated.

After all analysis we can say that using auxiliary arrays results in an improvement in the serial algorithm by reducing computation incurred in offset calculation while maintaining the same if not less cache miss rate. Removing these computation costs in the naive approach requires introduction of dependencies which renders the algorithm unparallelizable. Using auxiliary arrays has its own compute costs and overheads but doing so allows us to parallelise the problem and still get a considerable improvement over the naive approach.