Project Report

Comp 7850 - Advances in Parallel Computing

All Pair Shortest Path Algorithm – Parallel Implementation and Analysis

Inderjeet Singh

7667292

December 16, 2011

Abstract

There are many algorithms to find all pair shortest path. Most popular and efficient of them is Floyd Warshall algorithm. In this report the parallel version of the algorithm is presented considering the one dimensional row wise decomposition of the adjacency matrix. The algorithm is implemented with both MPI and OpenMP. From the results it is observed that parallel algorithm is considerably effective for large graph sizes and MPI implementation is better in terms of performance over OpenMP implementation of parallel algorithm.

1. Introduction

Finding the shortest path between two objects or all objects in a graph is a common task in solving many day to day and scientific problems. The algorithms for finding shortest path find their application in many fields such as social networks, bioinformatics, aviation, routing protocols, Google maps etc. Shortest path algorithms can be classified into two types: single source shortest paths and all pair shortest paths. There are many different algorithms for finding the all pair shortest paths. Some of them are Floyd-Warshall algorithm and Johnson's algorithm.

All pair shortest path algorithm which is also known as Floyd-Warshall algorithm was developed in 1962 by Robert Floyd [1]. This algorithm follows the methodology of the dynamic programming. The algorithm is used for graph analysis and finds the shortest paths (lengths) between all pair of vertices or nodes in a graph. The graph is a weighted directed graph with negative or positive edges. The algorithm is limited to only returning the shortest path lengths and does not return the actual shortest paths with names of nodes.

Sequential algorithm

```
for k = 0 to  N-1
          for i = 0 to N-1
                    for j = 0 to N-1

                              $l_{ij}(k+1) = \min(l_{ij}(k), l_{ik}(k) + l_{kj}(k))$
                    Endfor
          Endfor
endfor
```

The Floyd-Warshall Sequential algorithm [1]

Sequential pseudo-code of this algorithm is given above requires N^3 comparisons. For each value of k that is the count of inter mediatory nodes between node i and j the algorithm computes the distances between node i and j and for all k nodes between them and compares it with the distance between i and j with no inter mediatory nodes between them. It then considers the minimum distance among the two distances calculated above. This distance is the shortest distance between node i and j. The time complexity of the above algorithm is $\Theta(N^3)$. The space complexity of the algorithm is $\Theta(N^2)$. This algorithm requires the adjacency matrix as the input. Algorithm also incrementally improves an estimate on the shortest path between two nodes, until the path length is minimum.

In this project I have implemented the parallel version of all pair shortest path algorithm in both MPI and OpenMP. From the results I found that parallel version gave speedup benefits over sequential one, but these benefits are more observable for large datasets.

2. Problem and Solution

Parallelizing all pair shortest path algorithm is a challenging task considering the fact that the problem is dynamic in nature. The algorithm can be parallelized by considering the one dimensional row wise decomposition of the intermediate matrix I. This algorithm will allow the use of at most N processors. Each task will execute the parallel pseudo code stated below.

0	1	999	1	5
9	0	3	2	999
999	999	0	4	999
999	999	2	0	3
3	999	999	999	0

Table 1: 5 node graph representation with 5*5 Adjacency matrix. Nodes with no connection have weights 999

Parallel algorithm

```
for k = 0 to  N-1
        for i = local_i_start to local_i_end
                for j = 0 to N-1
                        Iij (k+1) = min (Iij(k), Iik(k) + Ikj(k))
                Endfor
        Endfor
endfor
```

The Floyd-Warshall Parallel algorithm [3]

Here local_i_start to local_i_end are the indexes decided by the partition size of the adjacency matrix i.e. value of N/P

In the kth step of the algorithm each task or processor requires in addition to its local data (bigger shaded row) the kth row of the same I matrix. Hence, the task currently holding the kth row should broadcast it to all other tasks. The communication can be performed by using a tree structure in log p steps. A total of N values (message length) is broadcasted N times. The time complexity will be the addition of computation and communication. The times complexity for will be $T_P = \frac{N^3}{P} + N\ logP(t_s + t_w N)$.

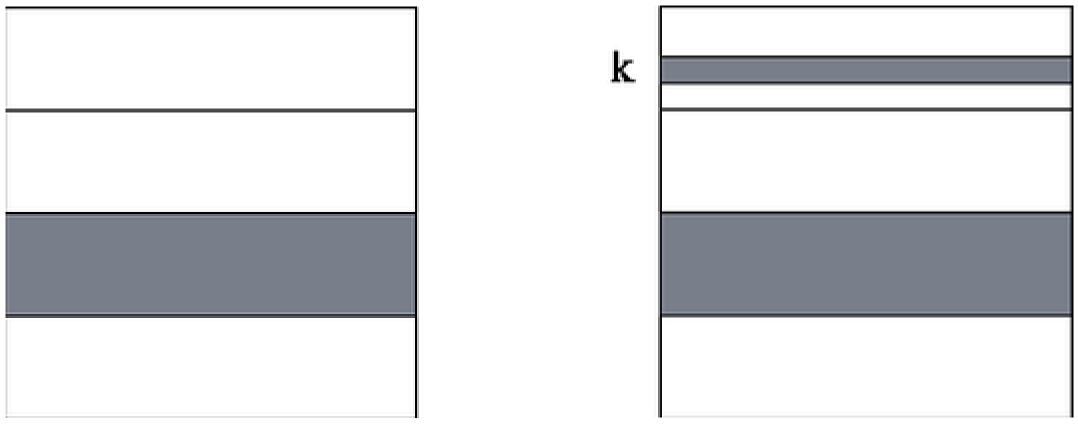

Figure 1: Parallel version of Floyd's algorithm based on a one-dimensional decomposition of the I matrix. In (a), the data allocated to a single task are shaded: a contiguous block of rows. In (b), the data required by this task in the kth step of the algorithm are shaded: its own block and the kth row ([3])

3. Implementation

For implementation I have both used MPI and OpenMP. Sequential and parallel (MPI and OpenMP) programs are written in c. For creating the adjacency matrix I have used the random number generator. Both the implementations of MPI and OpenMP works fine with the file as input. In terms of hardware setup Helium clusters in the computer science department of University of Manitoba are used.

In terms of number of nodes there are a total of six nodes, one head node and five computing nodes. The head node is a Sun Fire X4200 machine. It is powered by one dual-core AMD Opteron 252 2.6 GHz processor with 2GB RAM. In all the nodes Linux distribution of CentOS 5 is installed as and operating system. The other computing nodes feature 8 dual Core AMD Opteron 885 2.6GHz processors, making a total of 80 maximum cores. They all follow ccNUMA SMP (cache-coherent Non-Uniform Memory Access Symmetric Multiprocessing) computing model. Each computing node has 32 GB RAM size.

4. Results

The graphical results below give insights onto the execution time versus number of processes for different data sizes, speedup versus number of processes for fixed load size and efficiency versus number of processes for fixed load. Fig 2-4 shows the results for MPI implementation. Fig 5-7 shows the graph results for OpenMP implementation. Fig 8-9 shows the graph results for performance differences between OpenMP and MPI implementations.

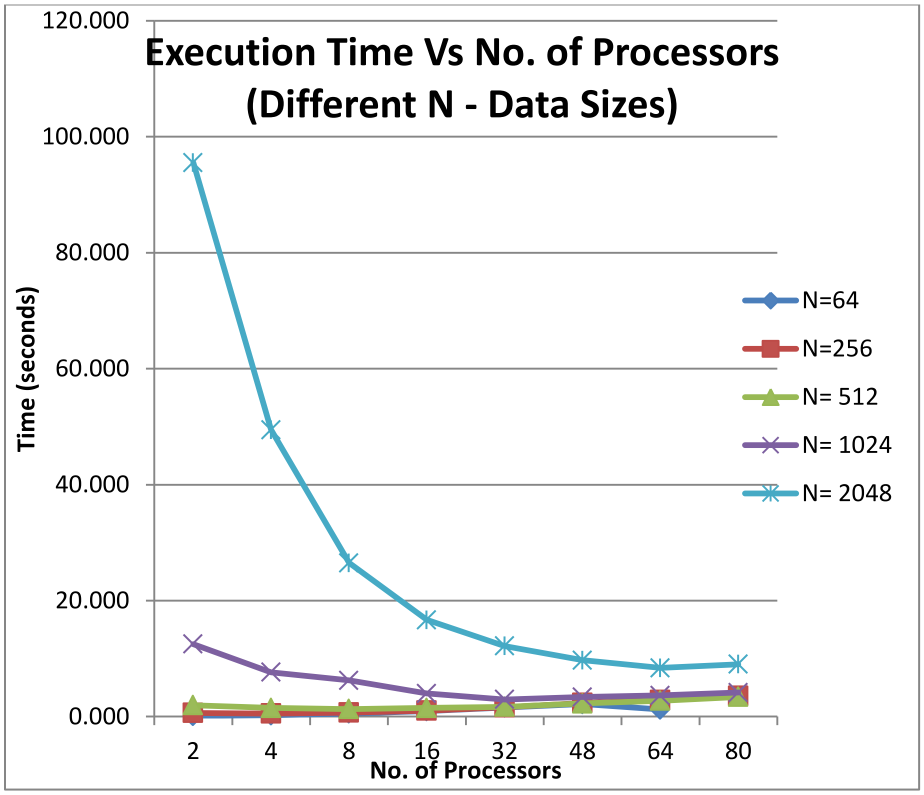

Figure 2: Execution Time Vs. No. of Processors for different data sizes

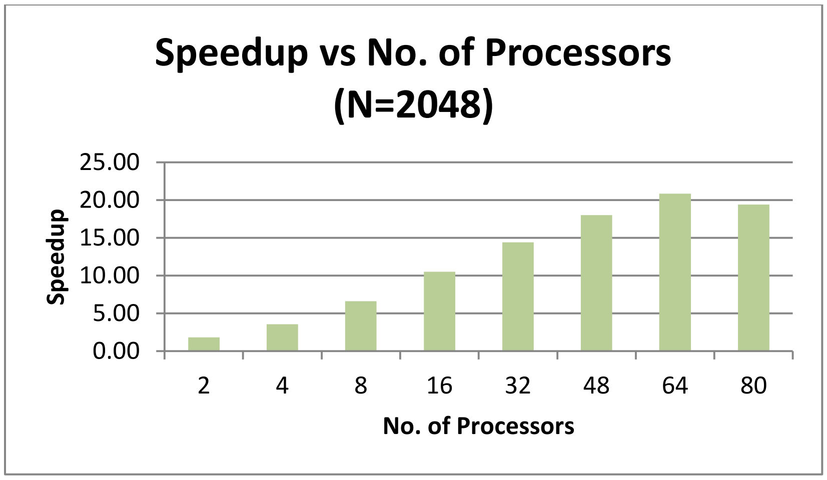

Figure 3: Speedup Vs. No. of Processors for node size of 2048

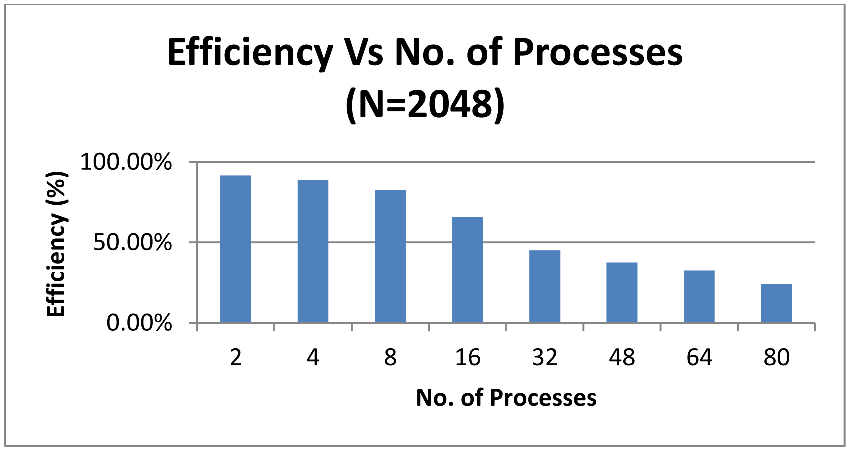

Figure 4: Efficiency Vs. No. of Processors for node size of 2048

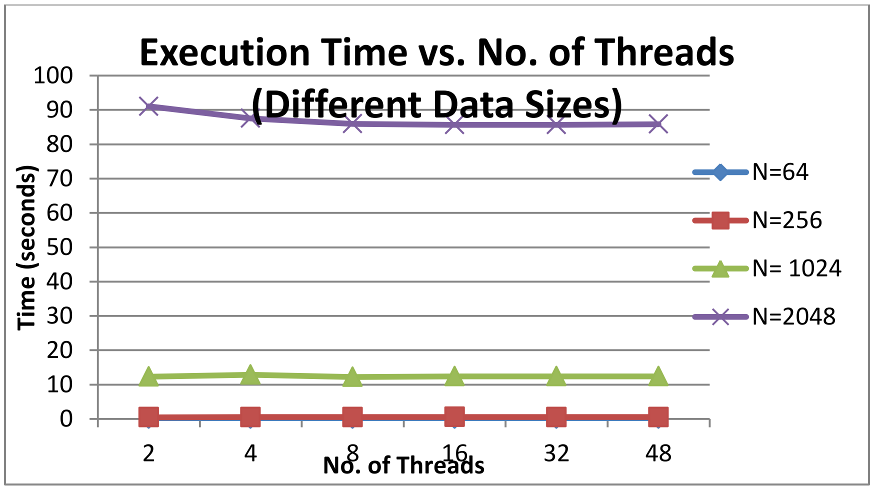

Figure 5: Execution Time Vs. No. of Threads for different graph size

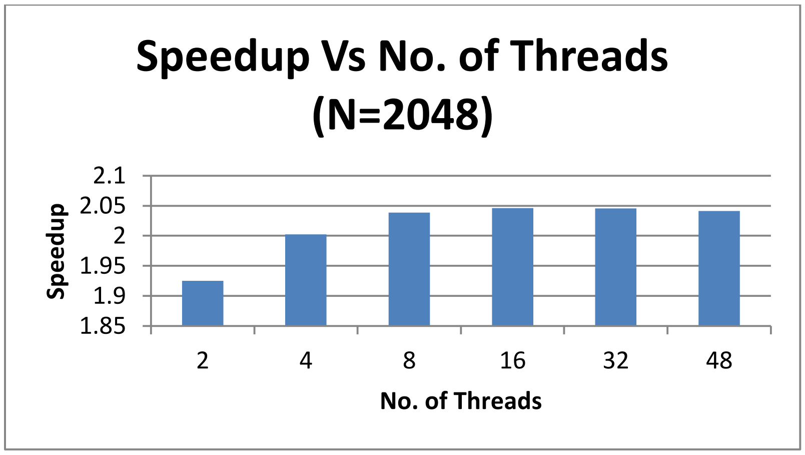

Figure 6: Speedup Vs. No. of Threads for node size of 2048

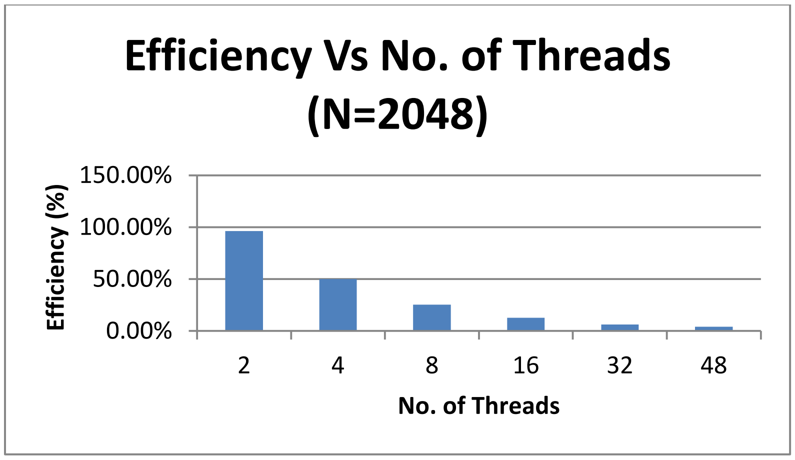

Figure 7: Efficiency Vs. No. of Threads for node size of 2048

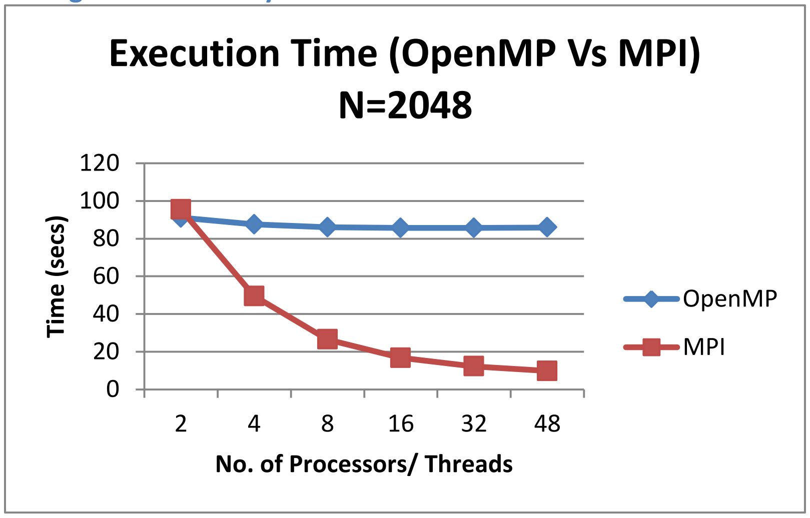

Figure 8: Execution Time Vs. No of Threads/Processes (OpenMP and MPI)

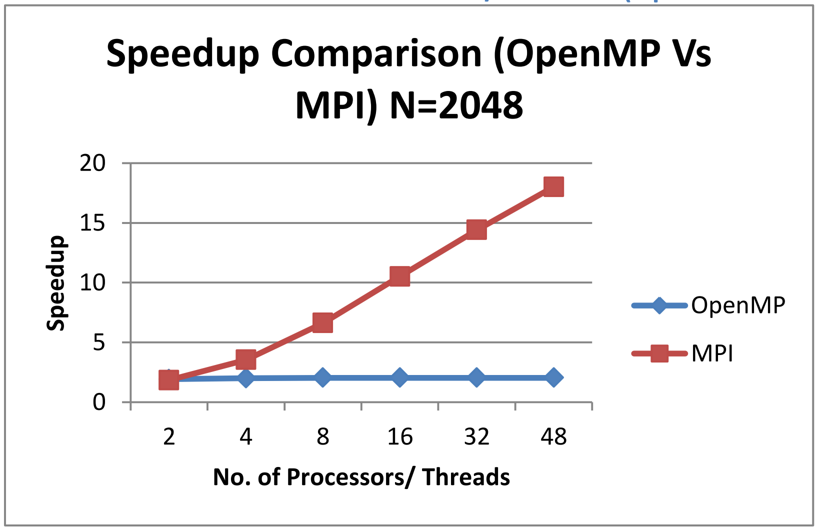

Figure 9: Speedup Vs. No of Threads/Processes (OpenMP and MPI)

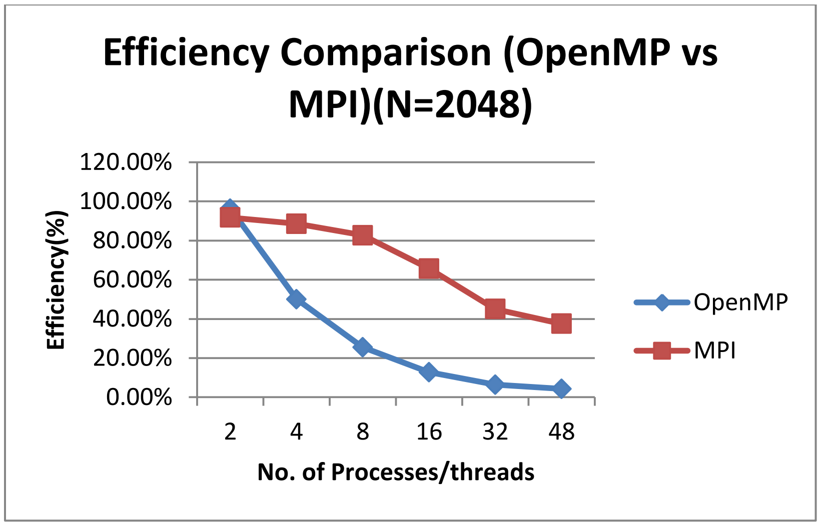

Figure 10: Efficiency Vs. No of Threads/Processes (OpenMP and MPI)

5. Evaluation and Analysis

As can be seen from the fig 2, that there is a sharp decline in execution time for node size of 2048 for increasing number of processes. The execution time almost reaches a constant value for increasing number of processes after 80 processes. There is a rise in execution time for small size of 256 nodes for increasing number of processes. This can be explained as the communication time overweighs the computation time for small node sizes. The data to be handled becomes more fragmented for small node sizes and more processes, hence more communication. From fig 3 it can be seen that for increasing number of processors speedup increases, but levels off towards the end for same node size. There is no speedup till there are 256 nodes. Fig 4 displays efficiency which becomes less for increasing number of processes for fixed load. This observation can be understood as more communication is required and more number of partitions of data is there.

From fig 5 it can be observed that execution time increases when the data size increases, but as the number of threads is increasing, instead of reduction in execution time, time becomes stable around 85 secs. From fig 6 it is observed that the maximum speedup is 2.04 when there are 32 threads for fixed 2048 number of nodes. Speedup increases in beginning but becomes stable at the end. For two threads there is maximum efficiency of 96.26% for dataset size of 2048, refer fig 7. Efficiency is only 4.25% for 48 threads.

While comparing the performance of MPI and OpenMP it is observed from graphs that MPI is more suitable for all pair shortest path algorithm. Fig 8 shows a clear difference between performances with respect to execution times. Speedup is close to two for OpenMP irrespective of the increase in number of threads, while for MPI speedup increases, refer fig 9. Efficiency wise OpenMP always performs badly for increasing number of threads, while MPI still maintains some efficiency for increasing processes. MPI is more efficient than OpenMP.

6. Conclusions and Future Work

From the observations I believe that the best use of parallel implementation of all pair shortest path algorithm is for large datasets or graphs. The speedups can only be observed with large adjacency matrices. From the results it is clear that MPI implementation if better than OpenMP for all pair shortest path algorithm. I think the reason can be because of distributed memory approach of MPI with local memories for the partitioned dataset, which are quite fast compared to shared memory access in case of OpenMP.

For future work I would like to do more analysis on the applications of parallel all pair shortest path algorithm on real life dataset such as that of social network that are publically available. I would also like to implement hybrid version of this algorithm.

7. References

1. Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM* 5, 6 (June 1962), 345-. DOI=10.1145/367766.368168 http://doi.acm.org/10.1145/367766.368168
2. Floyd-Warshall algorithm http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm, last accessed 30th, October, 2011
3. Case Study: Shortest-Path algorithms http://www.mcs.anl.gov/~itf/dbpp/text/node35.html#figgrfl3, last accessed 30th, October, 2011