PARALLEL ALGORITHM FOR FINDING BETWEENNESS CENTRALITY USING BRANDES ALGORITHM

Vijaya Karthik Kadirvel, Snehal Gandham

University of California, Irvine

ABSTRACT

In this project, We present the efficient parallel algorithm to find the betweenness centrality using brandes method. Brandes algorithm[1] presents the most efficient algorithm with time Complexity of O(nm) for unweighted graph and Space Complexity of O(n+m), where n is the number of vertices and m is the number of edges. Making it parallel reduces the complexity from O(nm) to O(nm/p) for an unweighted graph, where p is the number of processors.
We used two different parallel programming models, OpenMP and MPI to parallelize the algorithm. Our first algorithm uses a proper data-processor mapping, a novel edge-numbering strategy and a new triple array data structure recording the shortest path for eliminating conflicts to access the shared memory [2]. This can be done using OpenMP but this parallel model has few drawbacks which can be eliminated by MPI. Speedup achieved is significant using MPI rather than OpenMP.

Keywords -- Betweenness centrality, MPI, OpenMP, Shared memory model.

1. INTRODUCTION

Graph theory has wide applications in this present day world. In that betweenness centrality is a measure of a most influential node in the graph based on the shortest paths. For example, in a telecommunications network, Betweenness centrality is used to find the important node through which the information is passed and it points the node which has more control over the network. Betweenness centrality was devised as a general measure of centrality: it applies to a wide range of problems in network theory, including problems related to social networks, biology, transport and scientific cooperation[3].

The above problem can be approached in many different ways, Floyd Warshalls algorithm works with time complexity of $O(n^3)$ and has space complexity of $O(n^2)$ where n is the number of vertices present in the graph[2]. Another method of implementing is using Brandes efficient algorithm with the above mentioned time and space complexities. The last method is using Parallel Brandes algorithm which can produce results at a much faster rate than the previous two naïve algorithms.

The rest of the topics are organized as Section 2 describes the best sequential implementation of Brandes algorithm. Section 3 describes parallel brandes algorithm implementation using OpenMP and MPI along with the challenges and advantages of both the programming models. Section 4 discusses the results and findings that we obtained during our execution of the problem. Section 5 discusses the complexities that we achieved by using the parallel algorithm compared to the sequential algorithm.

2. SEQUENTIAL BRANDES ALGORITHM

The betweenness centrality is calculated by considering each vertex as the source vertex. Betweenness Centrality has been calculated in two stages,

In the first stage, the number of shortest paths and the predecessors for the vertex are stored. In the second stage, the betweenness centrality is calculated based on the shortest path ratios. Brande has assumed that vertex 'v' can lie on the shortest path between source vertex 's' and destination vertex 't' if and only if dG(s,t) = dG(s, v) + dG(v,t). Once the equation is satisfied shortest path is incremented by using the equation, sigma[neighbor] += sigma[vertex]. Also, the vertex is considered as the predecessor and added to the predecessor's list of the neighbor. Each of the vertices of the graph is pushed into the stack.

For the second stage, each vertex from the stack is used to calculate the Delta and Betweenness Centrality. Delta for each predecessor can be calculated by $\delta[v] += (\sigma[v]/\sigma[w])(1 + \delta[w])$, where $\delta[v]$ = shortest path ratio of predecessor, $\delta[w]$ = shortest path ratio of vertex of stack,$\sigma[v]$ = total number of shortest paths for predecessor v, $\sigma[w]$ = total number of shortest path of each vertex of stack. The betweenness centrality can be calculated using

the formula CB[w] = CB[w] + δ[w],where CB[w] = Betweenness Centrality for vertex of stack [1].

3. PARALLEL BRANDES ALGORITHM

In the parallel algorithm, we followed two different approaches, one using OpenMP and another using Message passing (MPI). The latter showed significant speedup when compared to the OpenMP parallel algorithm. We analyzed both the approaches and it helped us to understand which part of the program in OpenMP causes us the error and low speedup. The following helps you to understand our approaches much better,

3.1 Parallel Implementation using OpenMP

In this parallel implementation, The vertices are obtained from the input file. The input file contains vertices and its adjacent edges. In the sequential algorithm, we generally work on input vertices and based on that we will create the adjacency list. In parallel, We split the total number of vertices into individual groups of vertices for each processor. For example: if the number of vertices is 1000 and we are running in 10 processors, then each processor will have 100 vertices [2]. The group of vertices will be placed in a list for each processor and the vertices in the processors will work parallelly improving the parallelism. The queue is formed for each processor which holds the vertices which are visited for the first time [2]. The vertices in the queues are dequeued out parallelly to traverse the adjacency list and the adjacent vertices are enqueued in the corresponding queues.

In the sequential algorithm, we used predecessor set of the shortest path. We kept track of the predecessor set for each vertex to find the betweenness centrality of the graph. In the parallel algorithm, We are going to proceed in two stages,

First, we tried to use a new data structure called triple array [2] is used to store the parent, child, and level of a particular edge. Like the adjacency array, the triple array is shared by all processes and requires the handling of access conflicts [2]. In this case, each edge is assigned a unique number to assign in the triple array.

Second, the processor calculates an independent counter and keeps track of shortest path edges. Each processor extracts each vertex, neighbors of that vertex along with the level of neighbors. The individual processor will be storing the values in the triple array independently without any dependency. This parallel data collection of triple array reduces the BFS time we spent in sequential algorithm. The triple array is accessed in parallel to calculate the partial betweenness centrality of each vertex which is incremented from level to level. The summation of all the partial BC values will give the betweenness centrality of each vertex[2].

The sequential algorithm has a disadvantage of always having a dependency between two vertices in predecessor list which is not present in the parallel algorithm. Synchronization has to be carried out to make it work fine sequentially but this will not occur in parallel [2]. This algorithm executes the execution of triple array along with calculation of closeness parallelly but this is executed sequentially one after the another to produce the betweenness centrality.

3.1.1 Challenges Faced In This Parallel Algorithm

The bottleneck of this algorithm happens at the execution of each parallel functions which is sequentially executed one after the another i.e) the creation of vertex groups and calculation of closeness happens parallelly but one after the another. The calculation of betweenness centrality for each vertex in the input file is done sequentially after these parallel functions are executed. This creates a serious drop in the speedup created by the parallel algorithm used earlier. This is one of the major reason why we switched to another parallel execution model.

Another challenge, we faced in this algorithm is the parallel creation of threads using OpenMP. Using a fork() and join() execution model, the threads are created and joined based on the number of input threads. In our case, The threads are created based on input but parallelism is not achieved. This is another reason for choosing MPI over OpenMP.

3.1.2 Corrective measures for improving Parallelism

As discussed above, the two drawbacks made us think about another alternative parallel execution model. We tried to explore different parallel models and to figure out which will produce speedup with maximum parallelism eliminating drawbacks. The model that we have to choose for this particular problem should be showing the attributes like communication between two processors in order to calculate the betweenness centrality because this is one of the reasons why OpenMP fails to produce significant speedup compared to the sequential algorithm.
Another important aspect of the parallel execution model should be exploiting parallelism by creating a number of processes which produces speedup compared to sequential execution.
This two aspects of our problem made us choose MPI parallel execution model. In MPI, we use the

communicator world to communicate with processors using various built-in commands to send and receive data from processors. The parallelism can also be exploited by doing the parallel computation of closeness by various processors and the values are locally stored in the memory of that processor. So, MPI helps to eliminate both the constraints that we had for proper execution of our problem.

3.2 Parallel implementation using MPI

In this parallel implementation of our problem, We follow the algorithm similar to the sequential algorithm but rather than executing sequentially, We distributed load to various processors. The algorithm uses the built-in instructions of MPI to communicate with processors. Initially, The vertices are obtained from the input data set provided an adjacency list is formed for the vertices visited which is similar to the OpenMP algorithm. The parallel execution of the algorithm is carried out after the input vertices are read from input file. The MPI_BARRIER is called to create a synchronization between different processors executing the function. The adjacency list is created using BFS to add the vertices visited level by level for each processor.From the BFS execution, we calculated the closeness of all the vertices starting from the first vertex.The closeness values give the degree of closeness that particular vertex is present to its parent.This differs with the OpenMP implementation where we create a triple array with parent, child, and level <parent, child, level> [2].

After the closeness calculation, The Barrier helps to synchronize multiple processes to wait until it moves to the next phase of parallel execution. The most important aspect of our problem is communication. In OpenMP implementation, We used shared memory to hold the triple array which updates the array list parallelly by a different processor. This update causes bottleneck since the processor has to wait until another processor is updating. This bottleneck can be reduced by using MPI, where the array list and closeness calculation is done in each processor and stored in local memory.

During the betweenness calculation, The MPI_Reduce will calculate the betweenness centrality for all the vertices present in the input file. This calculation requires the processor to communicate with another to reduce and find betweenness centrality. The reduce instruction improves the speedup in MPI when compared to OpenMP where we have to access the triple array in order to calculate the betweenness centrality.

After the calculation of betweenness centrality, the output files are updated and the corresponding time for parallel execution is obtained. This parallel Implementation provides much better speedup when compared to OpenMP implementation.

3.2.2 Challenges faced in this implementation

The challenges that we faced during this implementation deals with the communication between different processors. The algorithm is not working fine when we tried to connect to the HPC queues. It displayed different errors in error log file generated after the program executes in the server. so, we tried to execute the MPI algorithm using mpirun instruction which creates the processes during run-time. we are not sure why this algorithm failed in HPC queue but this seems to be working fine on a server with mpirun instruction. the instruction creates processes and all the processes run the algorithm parallelly. this improves the speed up as we discussed in the previous sections.

4. RESULTS AND FINDINGS

The best sequential execution of Brandes algorithm produces the results in 3 min for 10000 vertices. In contrast to that, Parallel execution runs in 0.5 min. Further, The graph (figure 1) represents the time taken by both the sequential and parallel algorithm for the calculation of Betweenness Centrality for 10, 100, 4000, 10000 vertices respectively. There seems to be an exponential increase in the calculation time. MPI is used to parallelize the algorithm and decrease the calculation time. This creates significant change in the execution time.

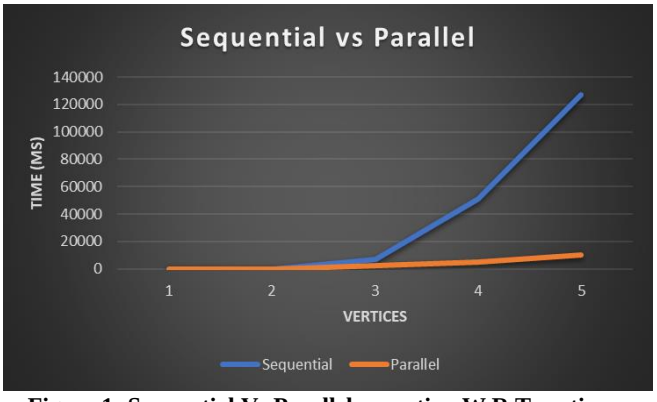

Figure 1: Sequential Vs Parallel execution W.R.T vertices.

Figure 2 represents the sequential and parallel execution of the algorithm w.r.t the number of edges. The graph gives

the execution time of both sequential and parallel time for different edges which depend on the density of graph [4].

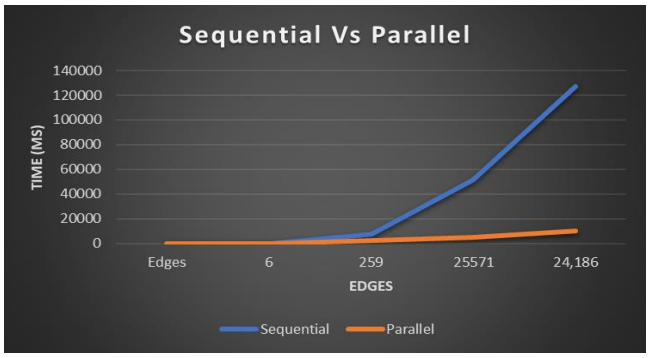

Figure 2: Sequential Vs parallel execution W.R.T edges.

Based on the above findings, we can say that the parallel algorithm executes at a faster time with respect to both edges and vertices. The following graphs give insight into the speedup achieved during the execution of the algorithm. The parallel algorithm will produce a speedup of 12x max for a large number of vertices this is described in figure 3 [4].

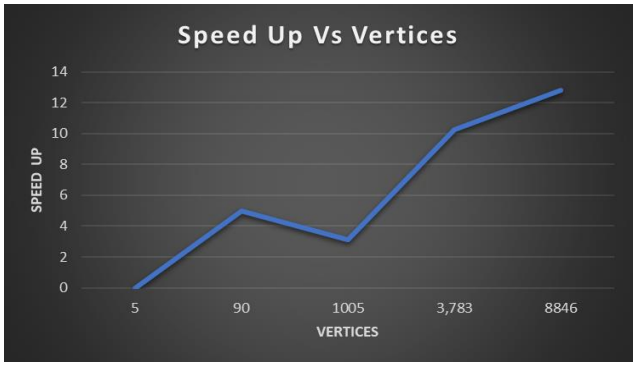

Figure 3: Speedup Vs Vertices

The speedup of parallel execution varies with respect to the edges which we plotted in the following graph. The speedup is significantly high with respect to a number of edges [4].

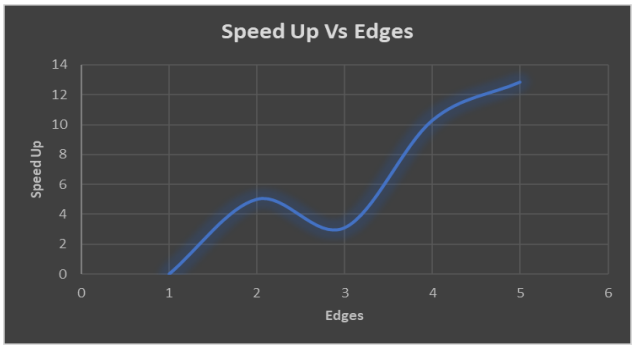

Figure 4: Speedup Vs Edges

The above graphs explain the findings we got during the execution of our sequential and parallel algorithm in the server. These findings infer that parallel algorithm runs at a faster rate than sequential making our problem solvable in O(mn/p).

5. COMPLEXITY ACHIEVED

The naïve sequential Floyd Warshall's algorithm can be achieved in $O(n^3)$ and space complexity of $O(n^2)$, where n is the number of vertices.The following best sequential brandes algorithm can be calculated in O(mn) and space complexity of O(m+n), where n is the number of vertices and m is the number of edges connecting different vertices.

The parallel brandes algorithm is executed with the complexity of O(mn/p), where n is the number of vertices, m is the number of edges connecting different vertices and p is the number of processors. This gives significant improvement in execution time and the corresponding speedup is achieved with respect to sequential algorithms.

6. REFERENCES

[1] Ulrik Brandes, "A Faster Algorithm for Betweenness Centrality" *The Journal for Mathematical Sociology*, pp. 163-177, 26th August 2010.

[2] Guangming Tan, Dengbiao Tu, Ninghui Sun, "A Parallel Algorithm for Computing Betweenness Centrality", *2009 International Conference on Parallel Processing*, IEEE, Vienna, pp. 163-177, 28th December 2009.

[3] https://www.geeksforgeeks.org/betweenness-centrality-centrality-measure/

[4] The datasets are taken from the following websites,

https://snap.stanford.edu/data/email-Eu-core.html

https://snap.stanford.edu/data/soc-sign-bitcoinalpha.html

https://snap.stanford.edu/data/p2p-Gnutella05.html