Scaling Graph Connectivity

Miriyala Jeevan Kumar (18111041) Nuka Siva Kumar (18111061)

I. Abstract

Graph connectivity is a fundamental problem in graph theory. It has wide range of applications in many areas in computer science for Eg. Computer Networks. Leader contraction algorithm is one of the popular solution for graph connectivity problem.It can be clearly observed that Leader Contraction algorithm can be parallelized as each leader contraction can happen parallelly. In this project we tried to scale the leader contraction algorithm to solve graph connectivity problem in parallel computing architecture using MPI.

II. Introduction

Graph Connectivity : Graph connectivity is the fundamental problem in the Computer science and mathematics fields. Two vertices are connected in an undirected graph if there is a path between them. A graph is connected if every pair of vertices is connected. [1]

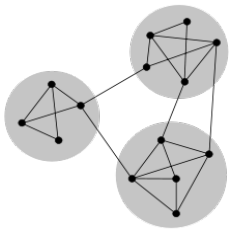


Fig. 1. This graph becomes disconnected when the right-most node in the gray area on the left is removed

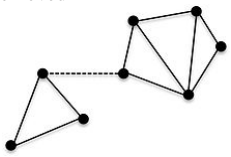


Fig. 2. This graph becomes disconnected when the dashed edge is removed.

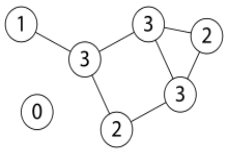


Fig. 3. With vertex 0, this graph is disconnected. The rest of the graph is connected.

MPI Collectives : In parallel computing architecture all processes will have individual data, for instance the result of a local computation, you may want to bring that information together, for instance to find the maximal computed value or the sum of all values. Conversely, sometimes one processor has information that needs to be shared with all. For this sort of operation, MPI has collectives. Following are some MPI_Collective calls which we have used in our project [2].

- **MPI_Bcast :** During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.
- **All_Reduce :** Many parallel applications will require accessing the reduced results across all processes rather than the root process, MPI_Allreduce will reduce the values and distribute the results to all processes.
- **MPI_AllGather :** Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes. In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast.
- **MPI_AllGatherv :**It gathers a variable amount of data from each process and sends the data to all processes. The MPI_Allgatherv function is like the MPI_Gatherv, except that all processes receive the result, instead of just the root.
- **MPI_Alltoall :** It gathers data from and scatters data to all processes. The MPI_Alltoall is an extension of the MPI_Allgather function. Each process sends distinct data to each of the receivers.
- **MPI_AlltoAallv :** It Gathers data from and scatters data to all members of a group. The MPI_Alltoallv function adds flexibility to the MPI_Alltoall function by allowing a varying count of data from each process.
- **MPI_Isend and Irecv :** These are are non blocking operations. Non blocking operations do not wait for any communication events to complete Non blocking send and receive: return almost immediately The user can modify a send [resp. receive] buffer only after send [resp. receive] is completed There are wait routines to figure out when a non blocking operation is done.

III. Leader Contraction Algorithm :

Leader contraction algorithm is one of the popular method for solving graph connectivity . It selects random set of vertices from all the vertices to be leaders, and for each non leader vertex it will check what are all the adjacent leader vertices. It will choose one of the leader vertex for contraction and contracts that non leader vertex to that leader vertex. The main challenge in this technique is to select a constant fraction of leaders that are adjacent to a constant fraction of non-leaders with high probability. Algorithm1 [3] contains the concurrent algorithm for it and repeat the same procedure till all edges in the graph has been contracted to their leaders and becomes

single vertex for each connected component in the graph.

Algorithm 1: Leader Contraction Algorithm :

Step 1 : Input G = (V,E)
Step 2 : Initialize i = 1, G_1 = (V_1, E_1) = G
Step 3 : In i-th iteration, if $E_i = \phi$, stop the procedure
Step 4 : Randomly choose some vertices from V_i as the leaders for G_i
Step 5 : For each non-leader vertex $v \in V_i$, if $\exists$ a leader u which is a neighbour of v in G_i, then contract v to u. (If there are multiple choices of u , choose an arbitrary one)
step 6 : Let $G_{i+1} = (V_{i+1}, E_{i+1})$ be the contracted graph and set i = i + 1
step 7 : At the end of procedure for each $v \in V$ if v is finally contracted to u then assign v the color of u.

IV. Implementation :

We have implemented leader contraction algorithm using various MPI_Collective calls. We have chosen the best Collective call possible for each communication required in the leader contraction algorithm. If we assume there are V vertices, E edges ,N processes, then our implementation of leader contraction algorithm is as follows.

- step1 : Distribute the edges and vertices among n processes which means each process will be assigned E/N edges and V/N vertices. We have used parallel IO read functions(MPI_File_read_at()) provided by MPICH for doing this.
- step2 : Each process will select leader from vertices assigned to it with a probability 0.5 for each vertex. It it will distribute this information among all processes .ie every process will gather leaders information from every other process, so that each process will know all the leaders which are selected in that iteration. We have used MPI_AllGatherv collective call for doing this. For generating random probability we feed the pseudo random function with process id, round no so that always it will generate different sequence probability.
- step3 : After gathering information about all leaders. Each process will select all possible contracting edges from its assigned edges and send that information to the corresponding to non leader adjacent vertex process. we have used MPI_Alltoall collective call to get Number of edges getting from each process then we have used MPI_Alltoallv collective call for doing this. While contracting the edges we must take care of which edges is to be contracted. If both ends are non-leaders we doesn't mark edge as contracting edge. If one end of edge has leader other is non-leader then mark that edge has possible contracting edge. If both ends are leaders then don't mark this edge as possible contracting edge.
- step4 : After step3 each process will have all the possible contracting edges with it for all its non leader vertices, now it will select one of the possible contracting edge and make that vertex points to the leader and send that information to all processes, so that each process will update their edges list according to updated contracted edge from all processes. we have used MPI_AllGatherv collective call for doing this.
- step5 : After updating the edges information .ie deleting the contracted edges, each process will now contain updated edges list, after this each process will get the total count of edges from all the process so that it can decide to proceed to next iteration or stop the procedure. Wee have used MPI_AllReduce collective call for doing this which will get the edge count from each process and do addition operation on it. While doing re links carefully checks the both ends of the edge and Contracted Node information. If both ends of the edge has been contracted then this edge will become edge between both the leaders. If one end of the edge contracted then edge now link to the leader of the contracted vertex. If both ends are not yet been contracted then edges become un link for this round. Before proceeding to next round all the processes will do MPI_Allreduce(SUM) on the edges left if all the edges are contracted then all processes will stop.
- step 6 : After Completion of all rounds Bases on the vertex contracted information we are giving color to the vertex. if a vertex not been contracted its color is it's vertex number only. If a vertex contracted to a leader Vertex both will get Color as final leader vertex number [remained single vertex]

V. Experimental Results :

Benchmarks : We have used graph benchmarks from Stanford Large Network Data Collection. Following are the large graphs on which we tested our code.

A. Wiki-Talk

wiki-talk contains the network of all the users and discussion from the inception of Wikipedia till January 2008. Nodes in the network represent Wikipedia users and edge from node i to node j represents that user i at least once edited a talk page of user j.This dataset has 23,94,385 nodes and 50,21,410 edges.

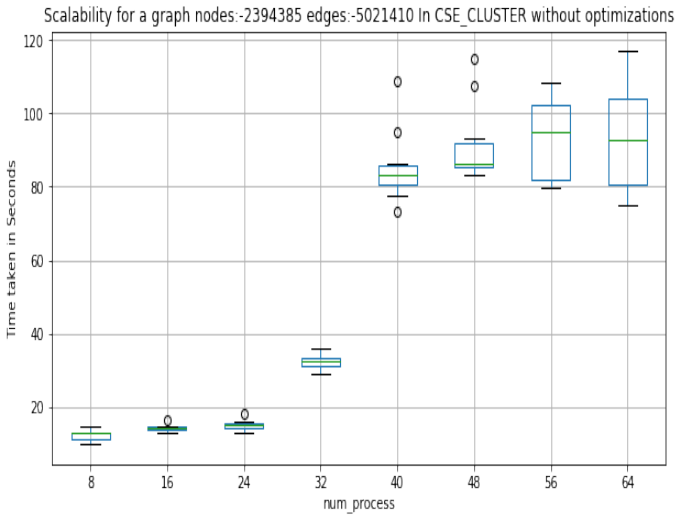


Fig. 4. Scalability for a wiki-talk graph In CSE_CLUSTER and ppn as 4 without optimizations

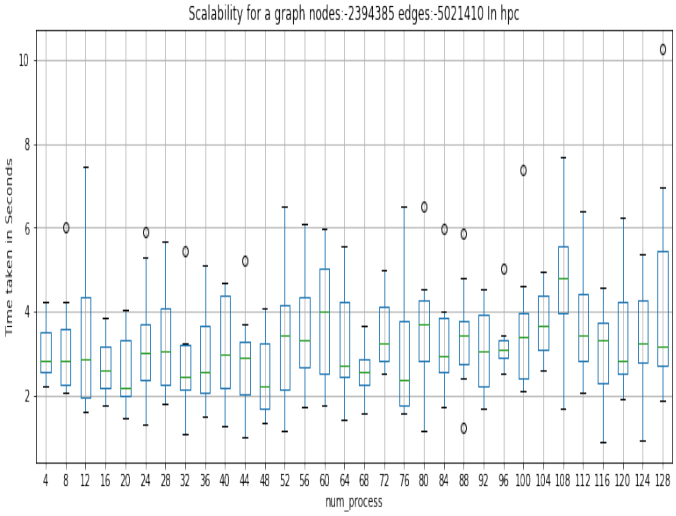


Fig. 5. Scalability for a wiki-talk graph In HPC without optimizations

- Settings of experiment : We graphed num of processes and their correspoing execution times in a boxplot with above mentioned Wiki-Talk benchmark. For every boxplot we have taken 10 readings.
- Observations : In CSE_cluster we can clearly observe the communication overhead while num_processes increasing then execution time is increasing.This odd behaviour is due to MPI_Collective calls like MPI_Allgatherv(), MPI_Alltoall(),MPI_Alltoallv(),MPI_Allredue(). In HPC environment we can't able to distinguish overheads between processes.
- Key Takeaways : Above observations clearly shows that our implementation is not well scaling so Improvement need to be done. First we have to find bottlenecks for the communication using pro filer. Later try to improvise the bottlenecks and check weather the implementation is well scaling or not. Later if any implementation overheads is there try to optimize those implementations example unnecessarily allocation, de-allocation of memory.

VI. Profiling

Profiling We need a instrumented toolkit for profiling and tracing for performance analysis of parallel programs. Profiling toolkit will give statistics about runtime environment of the parallel code.

A. TAU profiler

TAU [4] (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling.TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface.

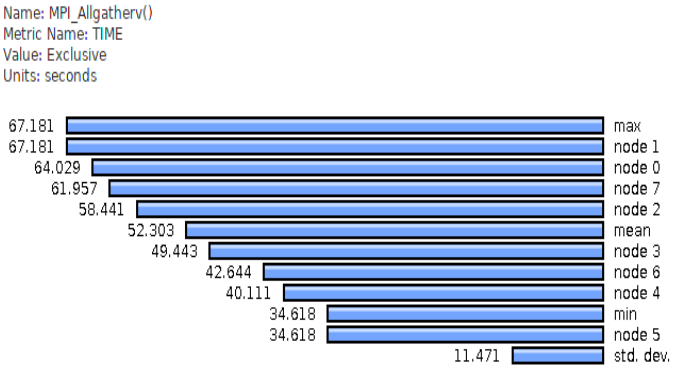


Fig. 6. MPI_Allgatherv() function profiler information without optimizations

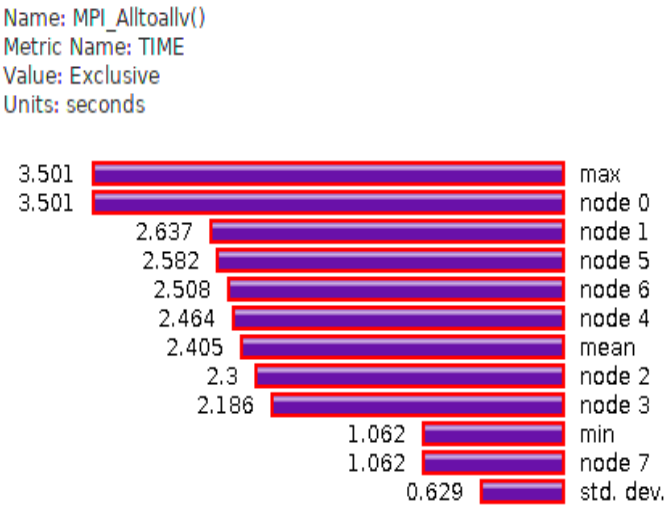
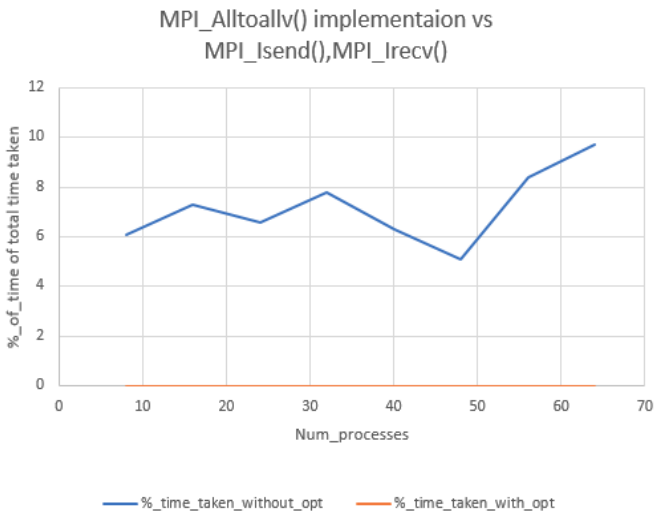


Fig. 7. MPI_Alltoallv() function profiler information without optimizations

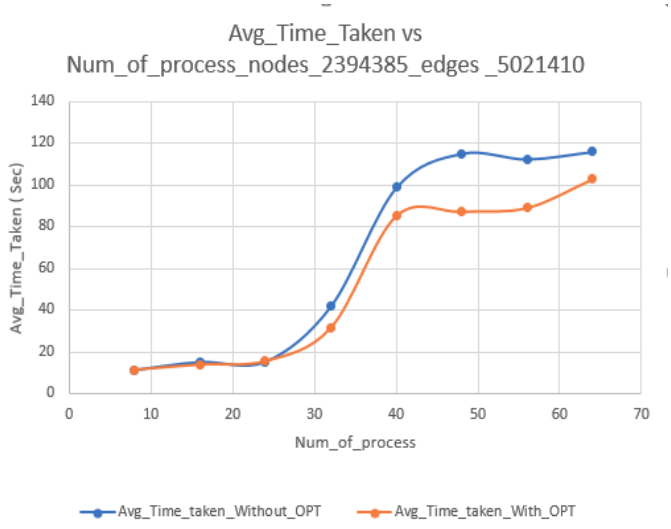
- Settings of Profiling : Fig 6 shows that TAU Visualization of total exclusive time of MPI_Allgatherv() in seconds at each node . Fig 7 shows that TAU Visualization total exclusive time of MPI_Aalltoallv() in seconds at each node . These results are taken from CSE_cluster
- Observations : We observed all the MPI_Calls and their percentage of time spent of total time. we figure that 65-80 % of the total is is spent by MPI_Allgatherv() function because. MPI_Allgatherv() function is used 2 times for every iteration one at knowing leaders information, another at contracting vertex information. About MPI_Alltoallv() is critical function in our this is function every node will send all the contracting edges information to every other function. It is taking 5-7 % of total time.
- Key Takeaways : Above observations clearly shows that our implementation is not well scaling so Improvement need to be done. We found that these functions can be improved . Later section we will compare our optimization results with In-optimized results.

VII. Optimizations

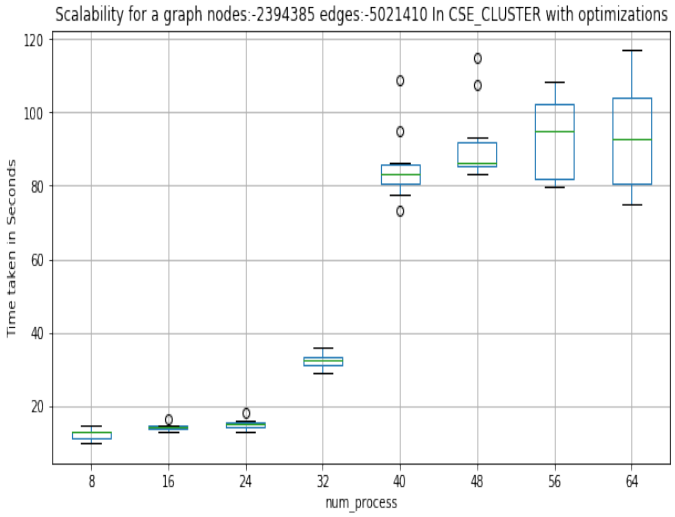
- Optimization-I : Our previous Implementation has more MPI_Barrier() calls this call will hold all the processes untill all other processes reached this point. If we remove Unnecessary Barrier calls it may improve. So we carefully checked all the code and figure it out what are the MPI_Barrier() calls can be removed so the correctness of the parallel code will not be distrubed.
- Optimization-II : In this Section we tried to remove blocking calls. By the Section-VI observation we found that MPI_Alltoallv() call is blocking call so It is taking 5-7 % of the total time. So We introduced Non-blocking calls that has correct implementation of MPI_Alltoallv() function. We removed MPI_Alltoallv() call and replaced with MPI_Isend() , MPI_Irecv() calls.
- Optimization-III : In this Section we tried to eliminate overhead of MPI_Allgatherv(). By the previous section we observed that 65-80 % of total is spent by this function. So this function has a scope to improvement. The problem with MPI_Allgatherv() is every process has its own sender buffer and receiver buffer . All the received data will be put at receiver buffer .So Here sender buffer is copied 2 times. we have optimized 2 times copy by introducing inplace copy of send buffer . It is already implemented in MPICH. So we introduced new flag called MPI_IN_PLACE. We copied send buffer directly into receiver buffer with respected position it reside
- Optimization- IV : Later we figure it out that we are allocation some buffers for each round and deallocating all the buffers at the end of the round. Round means In Implementation section Step-1 to Step-5 of parallel code path. We carefully Limit the allocation , deallocations in the code path. We can't able to completely remove all the allocation, deallocations because we may need a buffer that has larger size that previous round this applies recursively for every round. So we can able to avoid all the buffers deallocation except 1 buffer.



- Results : From the Fig. 8 we can clearly shows that our optimization has improved the performance. We observe that our optimization has improved because before MPI_Alltoallv() contributed 6-8 % of total time now our MPI_Isend, MPI_Irecv been contributing less than 1 %



- Results : From the Fig. 9 we can clearly shows that our optimization has improved the performance. We observe that if we increasing the num of processes the improvement has become more.



- Results : From the Fig. 10 we can

VIII. Future Work

We have only implemented the leader contraction algorithm in our project because of time constraint. But we can still improve the results by doing the densification of graph which will help in reducing number of rounds there by amount communication and computation required [3], we are leaving this part to future work.

IX. Conclusion

Graph connectivity is the fundamental problem, which can be parallized by using parallel computing tecniques. In this project we have tried to scale leader contraction algorithm which is a parallel graph connectivity algorithm using MPI, We have optimized our design by proposing time consuming Alltoall collective calls with non blocking sends and recieves.

X. Appendix

Member Contributions : We did not divide work between us, both of us have discussed and has done the work the together, its difficult distinguish the contribution of each person.

References

[1] wikipedia.org, “Graph Connectivity.”
[2] mpitutorial.com, “MPI collective communication.”
[3] Z. S. Z. W. P. Z. Alexandr Andoni, Clifford Stein, “Parallel Graph Connectivity in Log Diameter Rounds,” 2018.
[4] http://www.cs.uoregon.edu/research/tau/home.php, “tau.”