Parallel Programming (SE 295)
Assignment 02

Satish Kumar
M.Tech. SERC
Roll No. -11052

Problem 1

Programming Strategies:

1. Adjacency matrix obtained from data file **graph.dat** has been stored into CSR Format. For all directed edge of u to v, Collumn index stores 'v' nodes corresponding to row u, for all u. Also, I have not stored weights of edges because the PageRank algorithm doesn't need them.

2. Computations involved per iterations, primarily consist of two parts: 1. evaluation of **total sum** for each node (2,00,000 edge operations) 2. **Updating the rank** of each nodes (50,000 operations) from corresponding total sum value. Both the parts are highly independent in there computations, But part 2. must happen after part 1. Hence, Idea of parallelism is to divide the **part 1** computations equally among processes.

3. In order to balance the load across each processes, I used parallel binary search on row pointer vector, and then found out initial and final row(nodes from adjacency matrix) for each process, such that each process gets nearest of (2,00,000/nproc) of data to compute sum (part 1).

4. Also, Updating the page rank (part 2.) has been processed in parallel. Here, Each process updates rank for nodes/rows belonging to them as in part 1.

5. Since updating the page rank for a node requires total sum corresponding to that node, we must need to add local sum from each processes. But, Each process needs total sum only for nodes belong to them. MPI_Reduce has been used for this task.

Results: Data for Execution time in seconds, obtained for 30 iterations, are as follows:

np	1	16	32
Exec Time	0.2900	0.3600	0.8400
Speed Up	1.0000	0.8056	0.3452
Efficiency	1.0000	0.0625	0.0252

Plot Obtained are shown below:

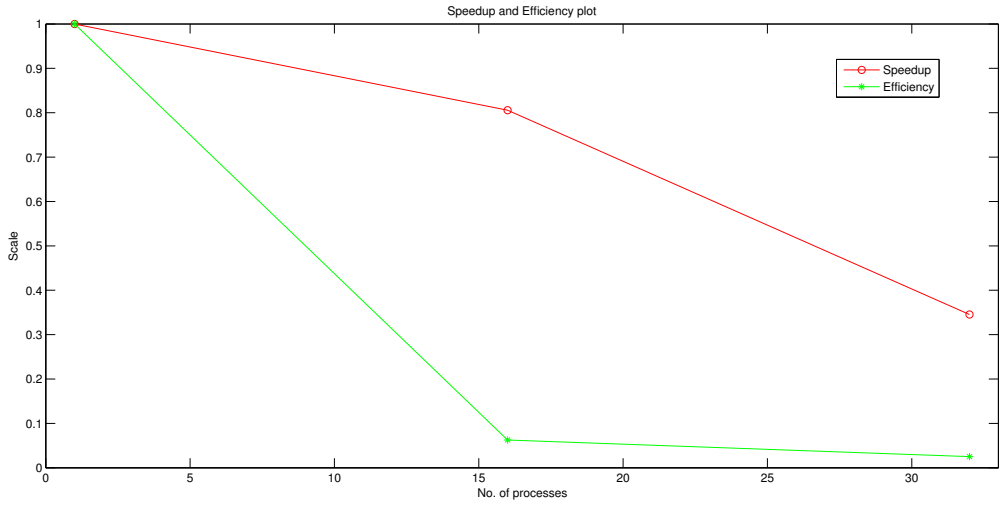

Fig. 1. Speedup and Efficiency Plot

Problem 1 continued on next page...

Observations:

- In sequential execution of the program, computations are linear in nature. Per Iterations, It does (2,00,000 * 1) mult/div for part 1 and (50,000 * 1) mult/div for part 2.
- In parallel program, MPI_Reduce functionality involves vector sum for np(16/32) vectors and then sends the sum to the relevant process. Hence, In each iterations, There are np such MPI_Reduce communication involved. When compared to MPI_Allreduce for communication, Sending the whole sum vector (50,000 x 1) seems unreasonable. Its a trade off between latency and bandwidth.
- Clearly, Reason behind speedup being less than 1, is due to communication being dominated over computations in each process. Profiling plot shows this quit clearly.
- For load balancing, Its difficult to balance load for both part 1. and part 2. evenly. Even if, we do that by having different distribution of nodes and edge among processes for part 1 and part 2 (i.e. even distribution of edge in part 1 and even distribution of nodes in part 2), We will require extra communications (i.e. communication of Page Rank) in each iterations.
- Even distribution of edge (as did in my program) is a arguably better load balancing keeping in mind of extra communication. Since, Part 1. does much more computations compared to part 2. In my program, Distribution of Rows/nodes and Collumn Index/edge (i.e. Row Pointer and Collumn Index) for each 16 process are as follows:

np	0	1	2	3	4	5	6	7
Initial Row	0	1027	2787	4786	7046	9590	12725	14945
Final Row	1026	2786	4785	7045	9589	12724	14944	18188
Col	12501	12500	12493	12495	12490	12492	12501	12497

np	8	9	10	11	12	13	14	15
Initial Row	18189	21877	25518	28070	31348	34820	38928	43752
Final Row	21876	25517	28069	31347	34819	38927	43751	49999
No. of edge	12508	122449	12487	12490	12478	12490	12490	12485

  The above table shows that edges (hence part 1.) has been almost equally distributed. But, Row distribution is not that much even (hence part 2.). Same pattern follows for 32 processes.
- the above table also shows that, In term of space, load balance is quite good.
- Hence, In term of computations, Even distribution of edge leads to load balance for part 1, but, part 2 computations may not be balanced always.
- In term of communications, Each process does np no. of MPI_Reduce, but size of array which are getting reduced differs. and, hence little imbalance happens there.
- But, Use of MPI_Allreduce or Different decomposition-assignment technique is not going to be helpful either in our test-case(graph.dat), since we have very less time of computations compared to communications.
- Since parallel computations will be helpful when we have a huge computations(lets say N=1e10 or more), for which communication delay is not highly dominating, Even distribution of edge would be the better choice of decomposition for load balancing.
- We can see this load imbalance in profiling screen shot as well(explained in next section).

Problem 1 continued on next page...

Profiling Outputs:

With the screen shots shown on the next page, we can see the time spent on MPI_Receive calls with time. Black portion shows computation time.

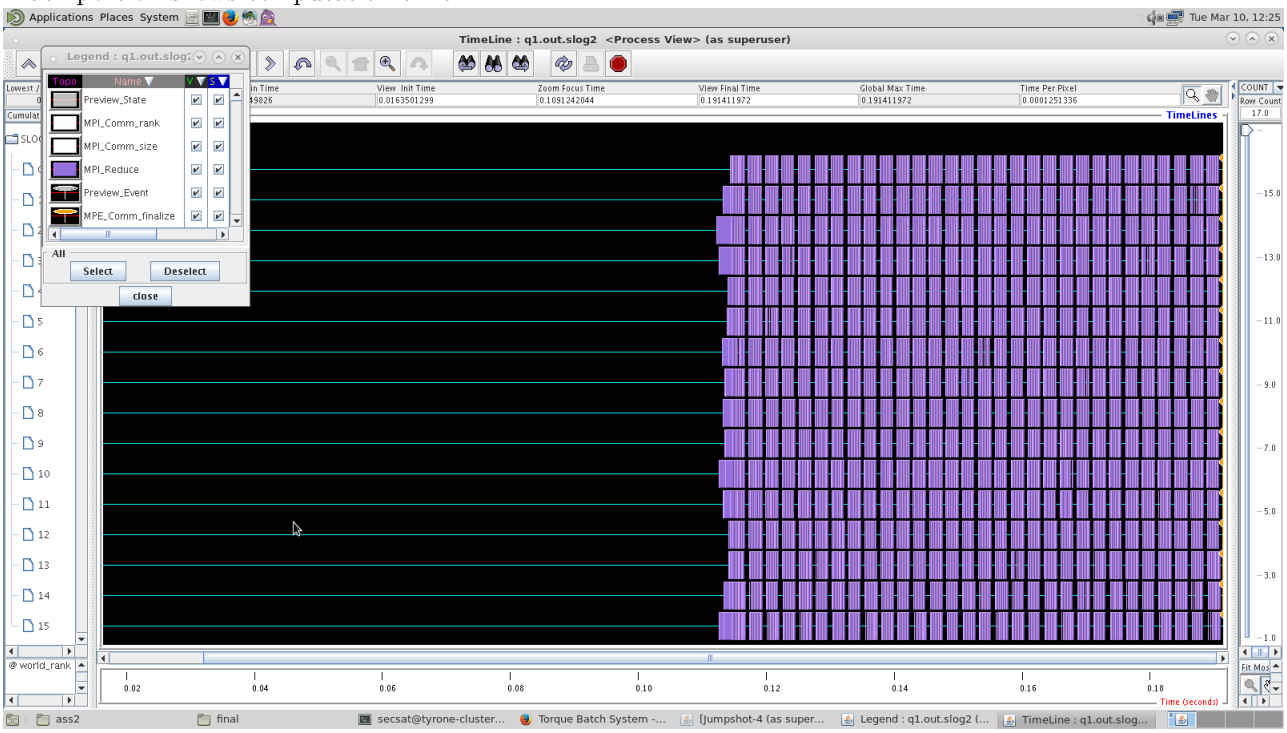

Fig. 2. Screen-shot of process view for 16 processes

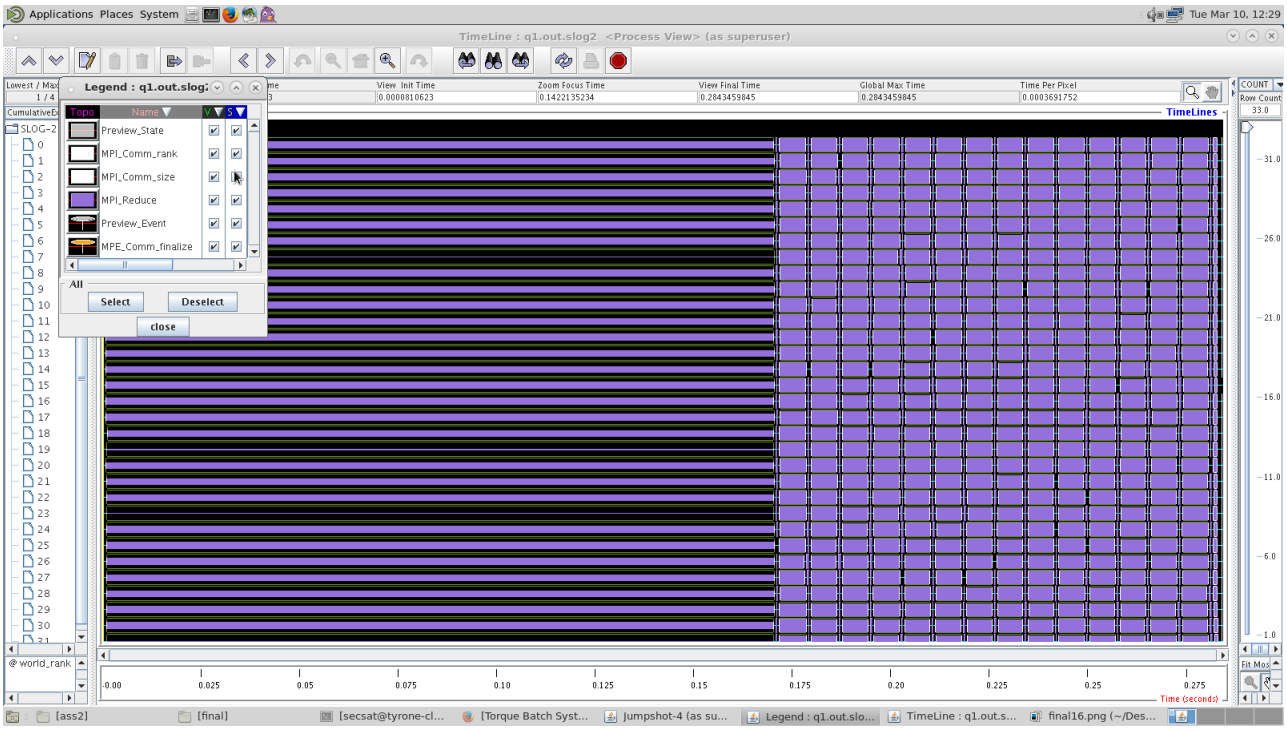

Fig. 3. Screen-shot of process view for 32 processes

Problem 1 continued on next page. . .

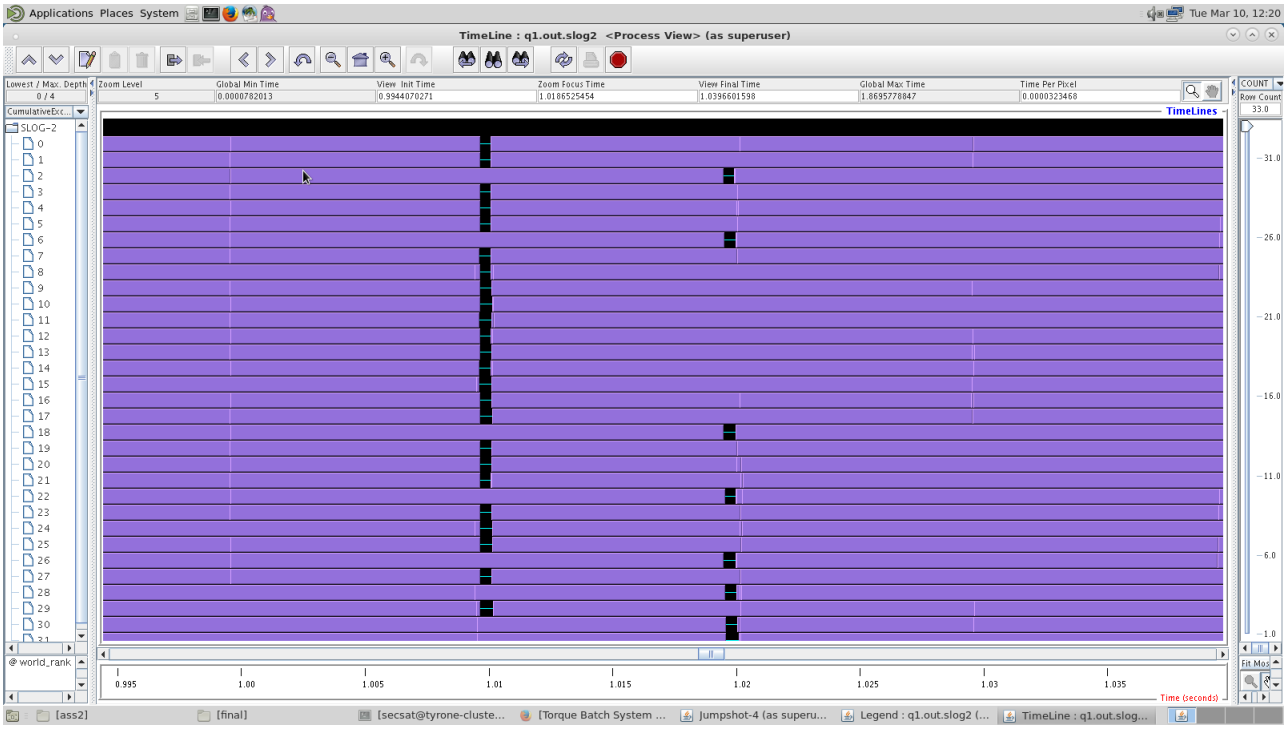

Fig. 4. zoomed process view

For load imbalance, we can see different length of time spent on MPI_Reduce by different processes. Also, Fig. 4 shows Computations which has been hindered by communication in Fig. 2. and Fig. 3.

Problem 2

Test Case and Strategies: To perform the analysis, I took A[12800][12800], x[12800]. and b = A*x.

Also, For wrap experiment, I choosed **NumBlocks = 50, NumThreads = 256** for fully populated wrap. And, **NumBlocks = 128, NumThreads = 100** for incompletly populated wrap in blocks.

Also, To perform coalesced experiment, I used the idea that, In a 2D array, data are stored in memory in row major fashion conitgously. Hence, A row wise computations will be optimized in data fetching. So, 2 kernel function has been written to compute in two different scenario.

Performence Benifits: Time in micro seconds are as follows:
1. Wrap and Coalesced : 2.51
2. No Wrap and Coalesced : 3.60
3. Wrap and No Coalesced : 3.01
4. No Wrap and No Coalesced : 3.89

- Excpectation was that, using multiples of wrap size as no. of threads per block, will optimize the instruction fetching, and Hence computations will be faster. Thats what we have observed from the experiment on test case.
- Expection was that, using coalesced memory access will optimize the data fetching and hence, computations will be faster. Same has been observed from the experiment on test case.

Note: I was unable to generate csv file to perform profiling.