CSCi 5451

Venkat Parthasarathy

6 May 2019

1 Description

1.1 Parallel Algorithm

1. Distribute to each process a chunk of vertices ($\approx n/p$) and their adjacency lists.
2. At each iteration of the page rank algorithm, each vertex pushes its page rank value and the other relevant information to the vertices connected by outgoing edges from it.
3. After all vertices perform the above operation, each vertex updates its page rank value.
4. The changed norm is first computed locally at each process and then reduced and broadcasted so that each process can break if the changed norm is less than the threshold.
5. Repeat steps 2-4 until we break/reach maximum number of iterations.

1.2 Implementation Details

1. Process 0 reads the file in chunks and does an `MPI_Send` of the chunk to the corresponding process.
2. Every vertex's adjacency list is read, the process at which the neighboring vertex resides is computed and the amount of data that each process needs to send to every other process is computed.
3. There is a global p x p matrix to which each process contributes a row. The entry p_{ij} indicates the amount of data process i needs to *send* to process j.
4. A `MPI_AlltoAll` is performed on this matrix to get the amount of data that process i needs to *receive* from process j.
5. Now, each process knows how much it needs to send and receive from every other process.

6. At every iteration of the page rank algorithm, a `MPI_AlltoAllv` is performed on the page rank values using the meta data computed in the previous steps. (Additional information regarding index management is also sent using another `MPI_AlltoAllv`)

7. Update Page Rank values from the received data.

8. The local norm is computed. Then a `MPI_AllReduce` is performed to find global norm and check whether we need to break.

1.3 Complexity Analysis

The complexity for each step in the "Implementation Details" section is provided: (V indicates the total number of vertices, E indicates the total number of edges)

1. $\mathcal{O}(V + E)$ = Each vertex and edge is read and sent once.

2. $\mathcal{O}(V/p + E)$ = Maximum of size of each chunk and number of edges processed. Worst case all edges can belong to a single process.

3. This matrix is filled out in the previous step itself.

4. A `MPI_AlltoAll` which is an All-to-All personalized takes time $t_s(p-1) + t_w.c.(p-1)$ where c indicates the size of message for an integer. Can be approximated as $\mathcal{O}(p)$.

5. N/A

6. Similar to the complexity analysis for step-4 but message size is no longer a constant. Can be upper bounded by saying each process needs to send every other process all its pagerank values. Time becomes $t_s(p-1) + t_w.(V^2/p).c.(p-1)$ if every vertex in the process needs to send page rank values to every other vertex in the graph (and thereby every other process). Can be approximated as $\mathcal{O}(V^2)$ as usually $p << V$.

7. $\mathcal{O}(V^2/p)$ This happens when every vertex in a process gets edges from every other vertex in the graph. Hence, updating page rank values for each vertex in the process becomes in order of V and quadratic if we consider all vertices in that process.

8. $\mathcal{O}(V/p + \log p)$ Computing local norm change takes $\mathcal{O}(V/p)$ and doing the all-reduce operation on one floating point number takes $t_s + t_w.c.\log p$ time which can be approximated as $O(\log p)$.

Hence, we can see that the complexity is dominated by step-6 which takes $\mathcal{O}(V^2)$ in the worst case. If we do the above steps for t iterations, the complexity becomes $\mathcal{O}(t * V^2)$

1.4 Parallel Overheads

- Filling out the p x p matrix.
- Performing the `MPI_AlltoAll` in step-4
- Performing the `MPI_AlltoAllv` in step-6
- Performing the `MPI_AllReduce` in step-8

1.5 Is your algorithm ideally suited for some types of graphs but not others?

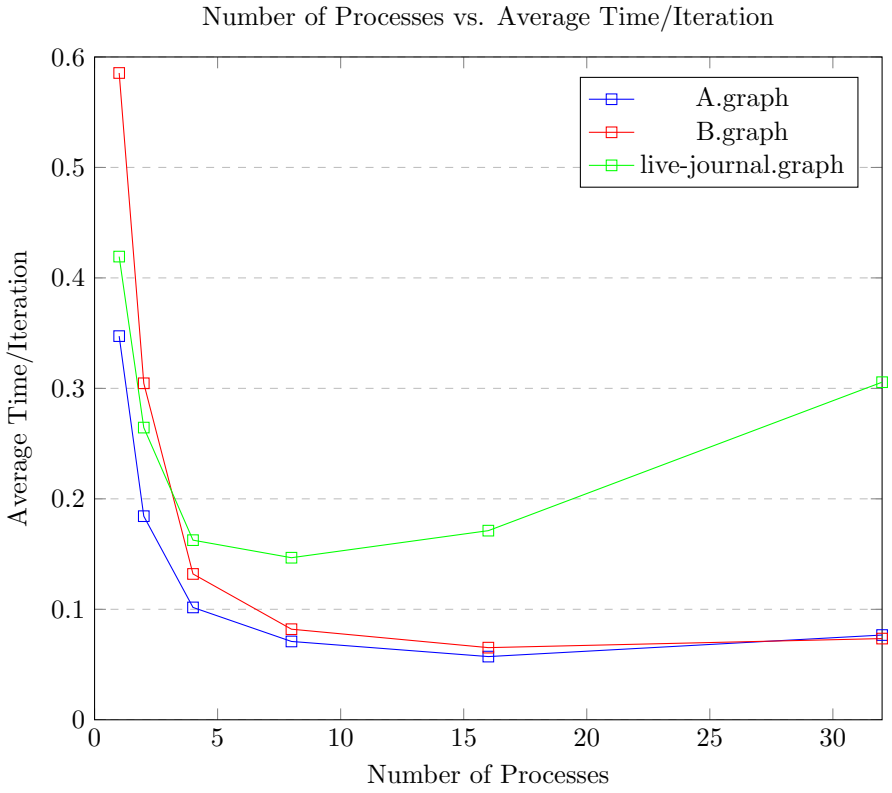
1. Algorithm is ideally suited for graphs where the number of edges per vertex is much smaller when compared to the number of vertices. This will help manage the complexity in step-6 and step-7 from quadratic ($\mathcal{O}(V^2)$) to close to linear ($\mathcal{O}(V)$).
2. It also works better if the number of edges per vertex is similar. This will ensure that there is not much idling where a process waits for another process to complete processing its chunk of vertices.
3. A back of the envelope calculation that might help us is to compare the number of edges with the number of vertices. If $E \approx V^2$, that is the graph is leaning towards a complete graph, then the parallel overheads might make it more expensive than the serial algorithm.

2 Results

2.1 Timing results

Type	1 process	2 processes	4 processes	8 processes	16 processes	32 processes
A.graph	0.3473s	0.1843s	0.1017s	0.0709s	0.0572s	0.0767s
B.graph	0.5855s	0.3046s	0.1320s	0.0820s	0.0653s	0.0735s
live-journal.graph	0.4193s	0.2645s	0.1626s	0.1467s	0.1712s	0.3056s

Table 1: Average time per iteration over 100 iterations



2.2 Understanding the results

1. The first aspect that strikes out from the graph is the decreasing returns when we increase we the number of processes. In fact, the time taken increases when we increase the number of processes from 16 to 32 in all three cases. This demonstrates that parallel overheads play a huge role in the runtime of this algorithm.

2. `live-journal.graph` doesn't provide as much speedup as the other two when you increase the number of processes. This harks back to the second point in section 1.5. I noticed that the distribution of work between processes is more equitable in `A.graph` and `B-graph` then in live-journal graph. This might also be the reason for the similar times even when you increase the number of processes.

3. The distribution of work is almost exactly the same in `A.graph` and `B-graph` but `A.graph` performs faster for almost all the processes. This is because of caching. If we observe successive adjacency lists in `A.graph`, they are almost the same except for one element. This helps in caching of those page rank values as the size occupied by each process on the cache is smaller. On the other hand, the adjacency lists in a chunk in `B-graph` almost do not overlap at all and hence more cache trashing happens than in `A-graph`.