15418 Final Project - Report

Arvind Mahankali (amahanka) and Clarissa Xu (csxu)

May 2022

1 Summary

We implemented Luby's algorithm for finding maximal independent sets in a graph, using MPI. Our implementation with 32 processes achieves a $21.5x$ speedup over a sequential version of Luby's algorithm, on a randomly generated graph with 2^{15} vertices and $100,000$ edges, and a $45x$ speedup on a randomly generated graph with 2^{15} vertices and $1,000,000$ edges.

Presentation Link

Presentation Slides

2 Background

Luby's algorithm is a parallel algorithm for finding maximal independent sets in a graph. The input to the algorithm is an adjacency list representation of the graph $G = (V, E)$, and the algorithm outputs a maximal independent set, that is a set S of vertices in G such that there is no edge in G between any two vertices in S, and for any other vertex $v \in V \setminus S$, there is an edge from some vertex in S to v. Figure 1 shows an example of all possible maximal independent sets for a given undirected graph.

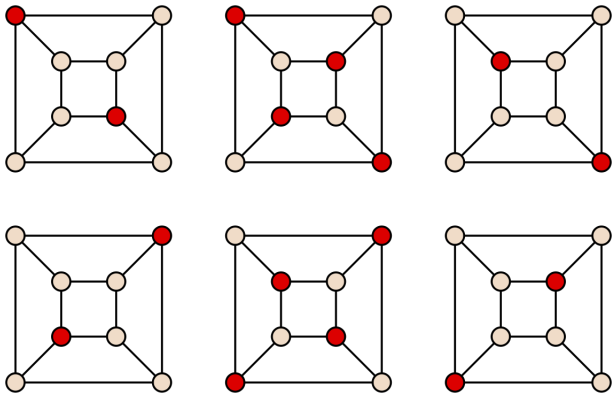
This algorithm maintains an independent set I which is grown over the course of several rounds. In each round, the algorithm maintains an active set A, which consists of all the vertices which could potentially be added to the independent set I — in other words, the vertices in G which have no edges to vertices in I.

In each round, a random priority is assigned to each vertex in A (this will be a uniformly random integer between 1 and n^c where c is chosen to be sufficiently large). For each vertex $v \in A$, if the priority of v is larger than the priorities of all of the neighbors of v which are also in the active set, then v is added to the independent set I, and all of its neighbors are removed from the active set A (they can no longer be added to I). The algorithm terminates when the active set A is empty.

Luby's algorithm can be broken down into the following steps:

1. Determining if there are any vertices left in the active set

Figure 1: Example of maximal independent sets (in red) (Source: https://commons.wikimedia.org/wiki/File:Cube-maximal-independence.svg (by David Eppstein))



2. Assigning random priorities to all of the vertices in the active set
3. For each vertex $v \in A$, going through the priorities of all the neighbors of v and finding the maximum of those priorities (and comparing this to the priority of v and determining if v should be added to the independent set)
4. Updating the independent set and removing the newly added vertices from the active set

Each step is dependent on the previous step. However, there is parallelism in the first, second and third steps — in particular, the third step is the most expensive. See [Ble] for a more detailed discussion of Luby's algorithm and its analysis.

A fast sequential algorithm for determining the maximal independent set is a greedy algorithm (which has been used by Blelloch, Fineman and Shun [BFS12] as a baseline). Pseudocode for the algorithm is shown in Figure 2 (this pseudocode was originally given by [BFS12]).

We implemented this algorithm and received the following thresholds:

Number of Vertices	Number of Edges	Runtime (s)
2^{15}	100,000	0.009453
2^{15}	1,000,000	0.020907
2^{15}	10,000,000	0.039592

We note that the max number of edges in a graph with 2^{15} vertices is $2^{15} * (2^{15} - 1)/2 = 536854528$ and we seek to choose test cases of varying densities.

We found our parallelism improvements to Luby's were unable to match the standard greedy algorithm, and thus, we implemented a sequential version of Luby's algorithm, removing the MPI overhead to reach the following thresholds:

Algorithm 1 Sequential greedy algorithm for maximal independent set

```
1: procedure SEQUENTIAL-GREEDY-MIS(G = (V, E), π )
2:     if |V| = 0 then return ∅
3:     else
4:         let v be the first vertex in V by the ordering π
5:         V' = V \ (v ∪ N(v))
6:         return v ∪ SEQUENTIAL-GREEDY-MIS(G[V'], π )
```

Figure 2: Sequential greedy algorithm pseudocode from [BFS12] (`https://arxiv.org/pdf/1202.3205.pdf`)

Number of Vertices	Number of Edges	Runtime (s)
2^{15}	100,000	1.191585
2^{15}	1,000,000	6.091636
2^{15}	10,000,000	71.919414

These benchmarks will be used for the speedup calculations to come.

3 Approach

Technologies Used: We implemented Luby's algorithm in MPI, and ran our implementation on the PSC machines with non-shared Regular Memory compute nodes.

Mapping our algorithm to parallel machines: We represented the vertices with the integers from 0 to $|V| - 1$. Let P be the number of processes we used. In our final implementation, we used a blocked assignment — in other words, we divided the vertices evenly between the processes, and the i^{th} process was responsible for the vertices $iK, iK + 1, \ldots, \min((i + 1)K - 1, |V| - 1)$.

Each process p maintains the following data, for each vertex $v \in V$ that is assigned to p:

- Whether v is in the active set
- If v is in the active set, then p also maintains the neighbors of v that are in the active set (importantly, these neighbors may not be assigned to p)
- The random priority assigned to v in the current round
- Whether v is in the independent set

Steps in our implementation: For the purpose of demonstration, consider Figure 3, an initial graph and accompanying adjacency list, with 8 nodes and 4 processes:

Our final implementation (which we refer to as V3) consists of the following steps, in each round:

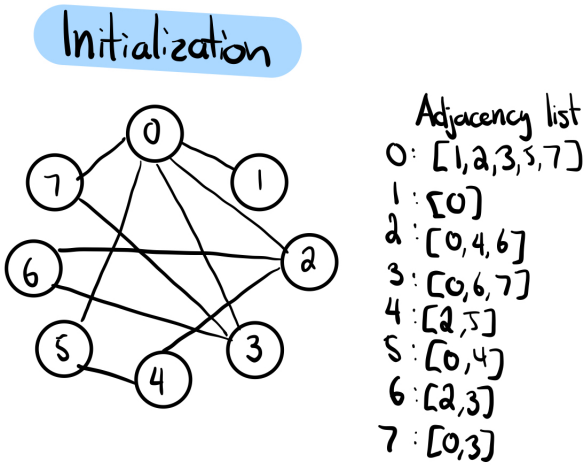


Figure 3: Initial Graph and Adjacency List)

1. First, the processes collectively determine the number of vertices in the active set — each process p counts the vertices assigned to p that are still in the active set, and the overall number of vertices in the active set is determined using `MPI_Reduce` and `MPI_Bcast`.

   For each process p, we also determine the other processes with which p will have to communicate with during this round — as will become clear later, p only has to communicate with process q such that an active vertex within p is connected to an active vertex within q. We do this simply by iterating through the active vertices v in p, and for each such v iterating through the neighbors w of v, and determining the process which w is assigned to.

2. Each process p sequentially assigns random priorities to each of the vertices assigned to it. Figure 4 shows this process.

3. Each process p determines which of the vertices assigned to p should be added to the independent set. In order to minimize communication, we divide this step into two substeps:

   - For each other process q, p first computes the message that should be sent to q. p iterates through the vertices v that are assigned to p. For each v, p then iterates through the neighbors of w that are still in the active set. If r_v denotes the priority of v, and q is the process that w is assigned to, then (w, v, r_v) is appended to the message that will eventually be sent to q. Figure 4 shows the accumulation of which nodes' priorities need to be sent to which processes. For each

process, we keep a vector containing the priorities to be sent and the nodes on that process who need to receive that priority.

- Next the processes communicate the relevant information with each other. Process p exchanges messages with process q for all processes q — if $p < q$, then p first sends its message to q and then receives q's message for p. In addition, p does not communicate with q if no vertex in p has any neighbors which are in the active set and are assigned to q (recall that we determine this in step 1). Figure 5 demonstrates a sample order in which priorities are sent. We use synchronous sends but it is not entirely sequential, as some exchanges can happen at once while other processes must wait their turn.

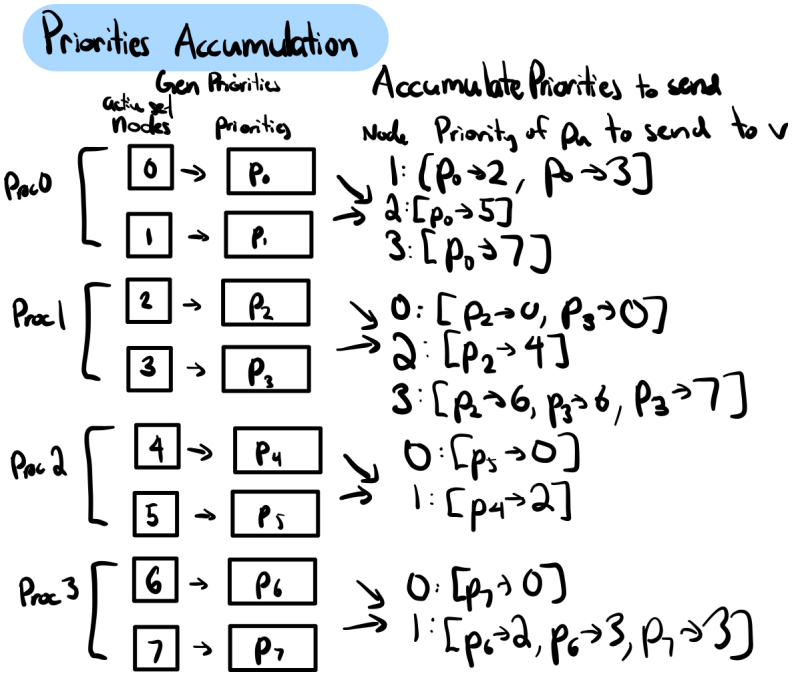


Figure 4: Accumulation of Priorities

4. At this point, for each vertex v assigned to process p, p has all of the priorities of all of the neighbors of v that are still in the active set. Thus, p can determine the vertices in its portion of the active set that have a greater priority than all of their active neighbors and can thus be added to the independent set. The first part of Figure 6 showcases this.

5. Now, all of the vertices which have newly been added to the independent set are communicated between all of the processes. Each process p sends its newly added vertices to process 0 using `MPI_Gatherv`, and process 0 sends

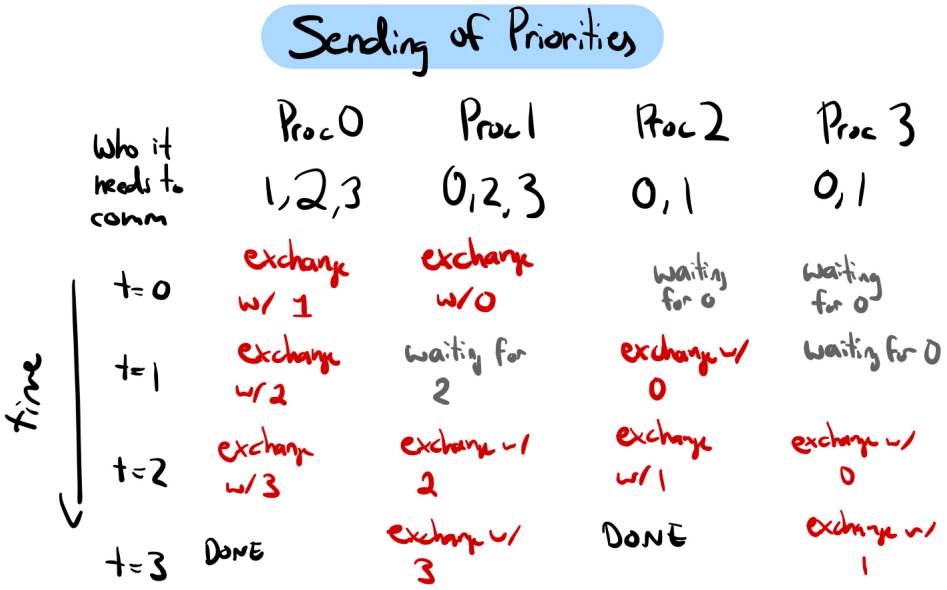


Figure 5: Sending of Priorities

all of the newly added vertices to all of the processes using `MPI_Bcast`. Figure 6 presents the Gather of the new independent set pieces from each process, followed by the broadcast to all processes.

6. Each process p must keep track of which of its vertices are in the active set. For each vertex v assigned to process p, p checks if v has any neighbor among the vertices newly added to the independent set — if so, v is removed from the active set.

7. Finally, note that each process p maintains, for each vertex v assigned to p, a list of the neighbors of v which are in the active set. We perform this in a similar manner as step 3 above — for each other process q, p first computes a message which contains all pairs (w, v) where v is a vertex on p that has been removed from the active set, and w is a neighbor of v on q. Then, for each other process q, if $p < q$ then p sends its message to q and then receives q's message for p (with the order being reversed if $p > q$).

3.1 Earlier Approaches

Initial Approach: In our initial implementation as well, we parallelized step 3 above, i.e. we parallelized the process of determining, for each vertex v in the active set, whether v should be added to the independent set. In our initial

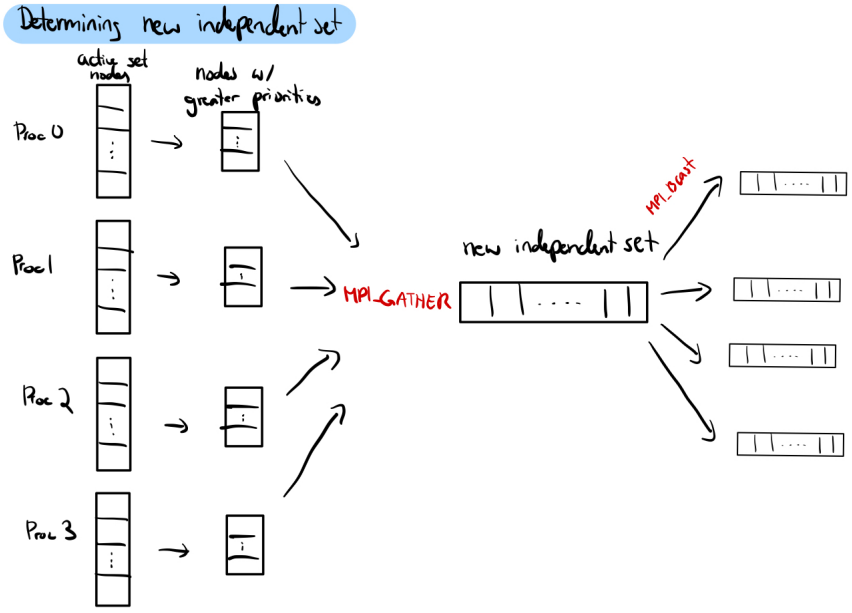


Figure 6: Creation of New Independent Set

implementation (which we refer to as V1 for convenience), all of the random priorities were shared between all of the processes. At the beginning of each round, the root determined all of the priorities for all of the vertices, and these random priorities were communicated to all of the other processes. Then, each process determines for a contiguous subset of the vertices whether those vertices have larger priorities than all of their neighbors.

All communication is facilitated at the root; the root determines the random priorities and broadcasts them, the root gathers all the maximal independent subsets and combines them and rebroadcasts them, and the root computes the new active set and broadcasts it as well. One potential disadvantage of this approach is that the entire independent set and active set, as well as the priorities for all of the vertices in the active set, are communicated to each process. The motivation for our next approach is that communicating to each process only the information that it needs (i.e. priorities of the neighbors of the vertices which are assigned to that process) may result in less communication, or at least the latency of communication even if it increases the total communication between all of the processes.

Luby's Algorithm — Blocked Assignment: Our next approach (which we refer to as V2) was similar to our final approach described above. Each process only maintains the information for the vertex that are assigned to it

(i.e. whether a vertex is active, whether it is in the independent set, which of its neighbors are active, and the priority of the vertex). The advantage of this is that it is not necessary to broadcast the random priorities to all of the processes. One crucial difference between this approach and our final implementation was how we implemented step 3 (where we wish to determine which vertices assigned to this process should be added to the independent set). In this approach, we implemented step 3 as follows:

- First, the process p collects all the edges (u, v) where u is assigned to p and v is a neighbor of u that is in the active set. p then sorts all of these edges (first by the smaller of u and v, then by the larger of u and v).
- Then p iterates through these edges in increasing the order, and for each edge (u, v) where u belongs to p and v belongs to another process q, p exchanges a message with q where p sends the priority of u and receives the priority of v.

Note that we avoid deadlock since we sorted the edges (u, v), meaning that all of the processes go through the edges according to the same order. However, the latency of communication of this approach is potentially high, since p might have to exchange a message with q once for an edge e_1, and then another message for a later edge e_2 — the process q might have to exchange messages with several other processes in the meantime. This motivated the following optimizations:

1. First, to determine the priorities of the neighbors of the vertices in its portion of the active set, process p instead iterates through all of the other processes q, and exchange a single message with q (consisting of all of the neighbors of p which are in the active set, together with their priorities and the vertices in p of which they are neighbors). This is similar to how we had already implemented step 7 in version 2.
2. Observe that in some cases, the above change might result in additional unnecessary communication, since q might have no neighbors with p, and this would lead to a message of length 0 — thus, the additional change we made to our implementation of step 3 is that if p has no neighbors in q which are in the active set (which can be determined beforehand) then p and q do not exchange a message. We performed this optimization for step 7 as well.

These were the optimizations we included in V3, and led to a significant improvement in speedup (relative to our sequential implementation of Luby's algorithm), as shown in Figure 7.

Other Optimizations: Another optimization we attempted to make to version 2 is as follows. Note that in our implementation of step 7 in version 2, each process p sends to process q a message consisting of pairs of the form (u, v) where u is a vertex in process p that is being removed from the active set and

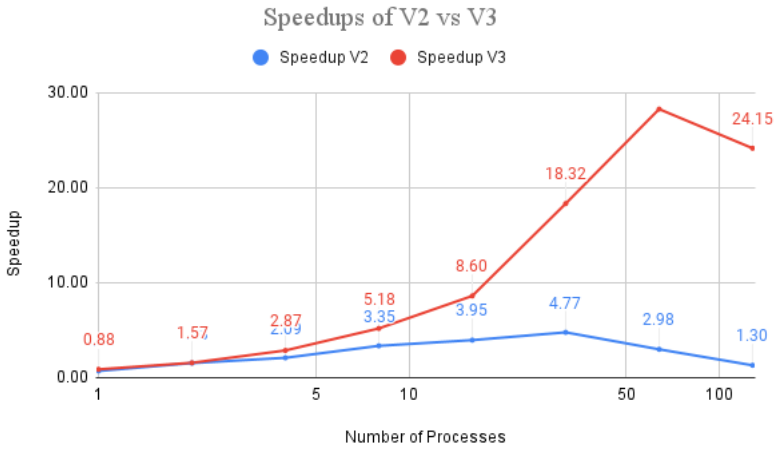


Figure 7: Speedup of V2 and V3 relative to our sequential implementation of Luby's algorithm

v is a neighbor of u in process q. However, note that only u needs to be communicated to process v, since process v can then iterate through its vertices in the active set to determine which vertices u is a neighbor of. Thus, we believed that if p only sent the us to q and q determined the v independently, the reduced communication would lead to better speedup. However, we found that on a graph with 2^{15} vertices and 100000 edges, this significantly worsened the performance with only 1 process, though the performance with larger number of processes was roughly the same as our previous implementation. It is likely that this proposed optimization does not lead to much improvement since the number of edges (u, v) between p and q, where u is being removed to the active set and v is in the active set, may be much lower than the overall number of edges between p and q.

4 Results

We measured performance in terms of wall-clock time, and speedup relative to a sequential version of Luby's algorithm (i.e. same as V2 but without any calls to MPI functions). We measured the performance of our algorithms on randomly generated graphs, which were created according to the procedure described in [CZF04]. We ran our final implementation with the following configurations:

1. 2^{15} vertices and $100,000$ edges
2. 2^{15} vertices and $1,000,000$ edges
3. 2^{15} vertices and $10,000,000$ edges

Running time and speedup plots for V3, for configuration 1, are shown in Figure 8. Plots obtained by running V3 on configuration 2 are shown in Figure 9.

Observe that with $100,000$ edges, the speedup is largest with 64 processes, and slightly less with 128 processes. However, with $1,000,000$ edges, the speedup is significantly higher with 128 processes than with 64 processes. Finally, with $10,000,000$ edges, the speedup is again slightly lower with 128 processes than with 64 processes, and furthermore, is lower overall with $10,000,000$ edges compared to with $100,000$ edges (for example, the speedup with 64 processes is now only $17.79x$ when the input graph has $10,000,000$ edges as opposed to $28.27x$ when the input graph has $100,000$ edges).

Our proposed explanation for this is that with $100,000$ edges, the problem size is too small for 128 processes to give a significant advantage over 64 processes — thus, communication overhead dominates for 128 processes. For 64 processes, the running times for each of the steps are shown in Table 1, while the time spent in each of the steps with 128 processes is shown in Table 2. With 64 processes, the percentage of time spent on step 6 (that is, updating the active set, which is by far the most expensive step with 1 process) is roughly 31%. However, while the time spent on this step with 128 processes is roughly halved, the time spent on step 3 and 5 (which consist mostly of communication) more than doubles. Thus, at this point, the communication overhead offsets the gains from dividing the workload among the processes.

Step	Running Time
1	0.008599
2	0.000129
3	0.005195
4	0.000269
5	0.005064
6	0.012450
7	0.007835

Table 1: Running time spent in each step of V3 with 64 processes on configuration 1 (note that here we report the times for process 0).

However, this is not the case with $1,000,000$ edges. For this case, we show the running time that V3 spends on each step with 64 (Table 3) and 128 (Table 4) processes. The main observation is that the time spent on steps 1, 5 and 7 decreases significantly. Note that in step 1, in addition to counting the vertices in the active set, each process p determines which other processes it needs to communicate with (i.e. other processes which contains vertices in the active set which are neighbors of vertices in p). By increasing the number of processes, we decrease the number of edges that each process has to consider — it is likely that this effect only becomes noticeable with $1,000,000$ edges. In addition, the speedup in step 7 is also likely because the size of the messages that each process has to send has decreased.

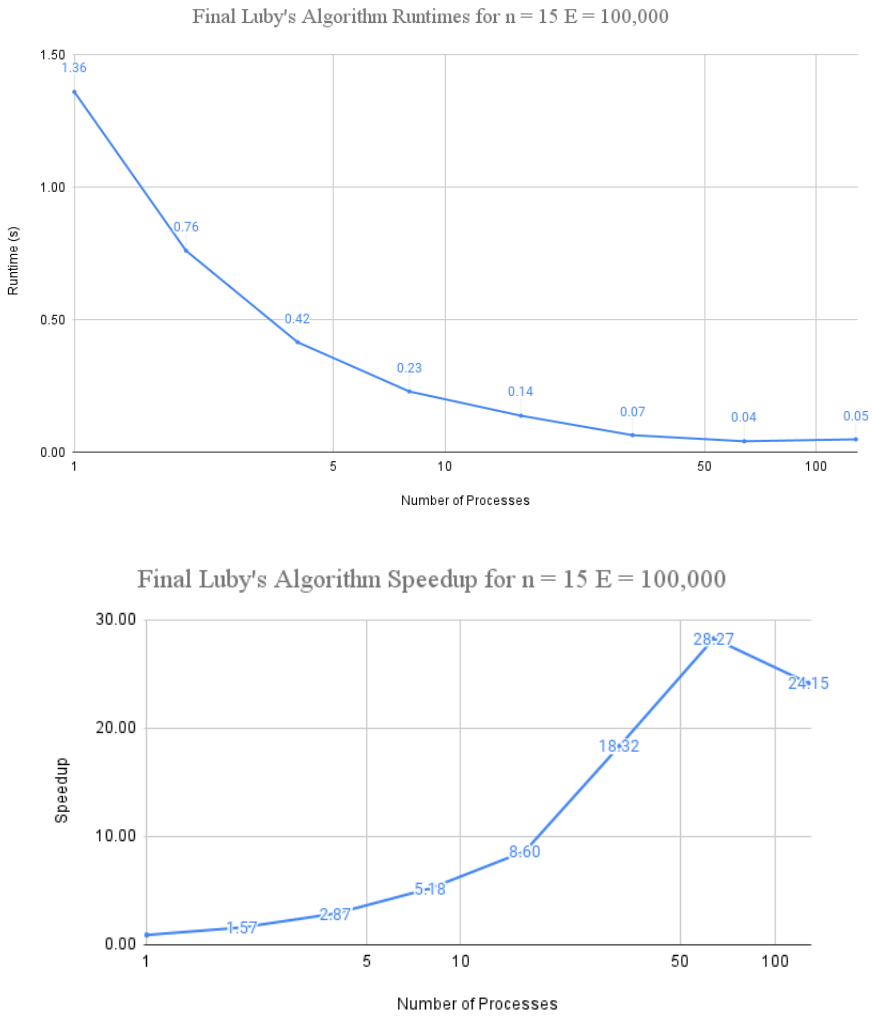


Figure 8: Running time (in seconds) and speedup (relative to our sequential implementation of Luby's algorithm) of V3 on configuration 1, with $1, 2, 4, 8, 16, 32, 64, 128$ processes

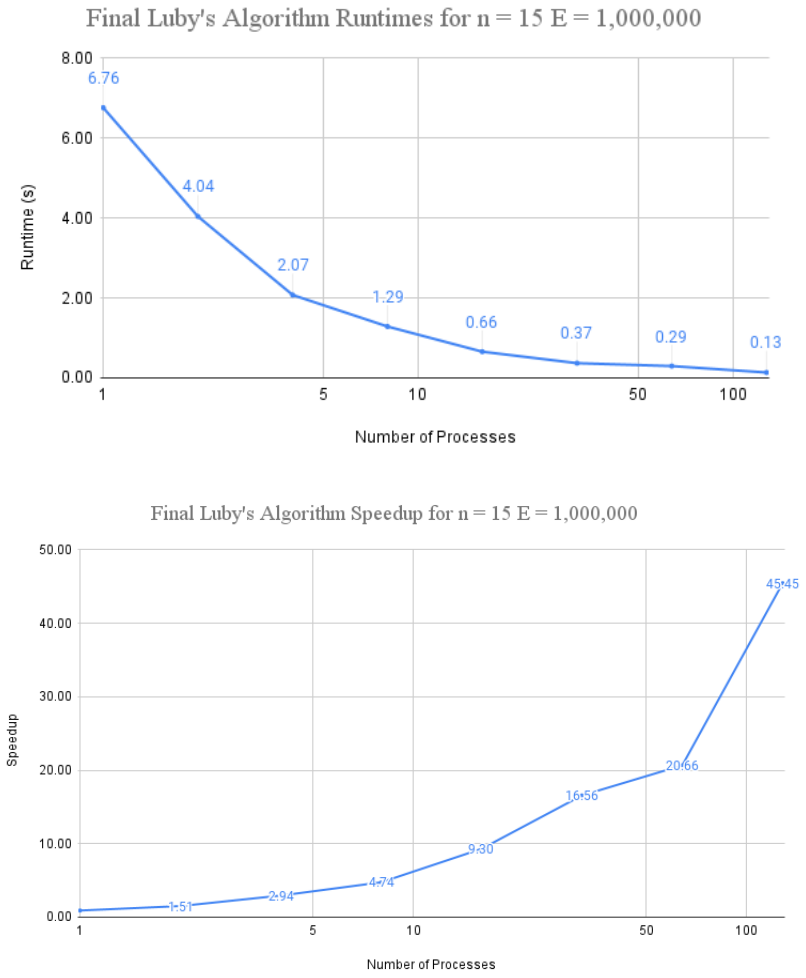


Figure 9: Running time and speedup of V3 on configuration 2, with $1, 2, 4, 8, 16, 32, 64, 128$ processes

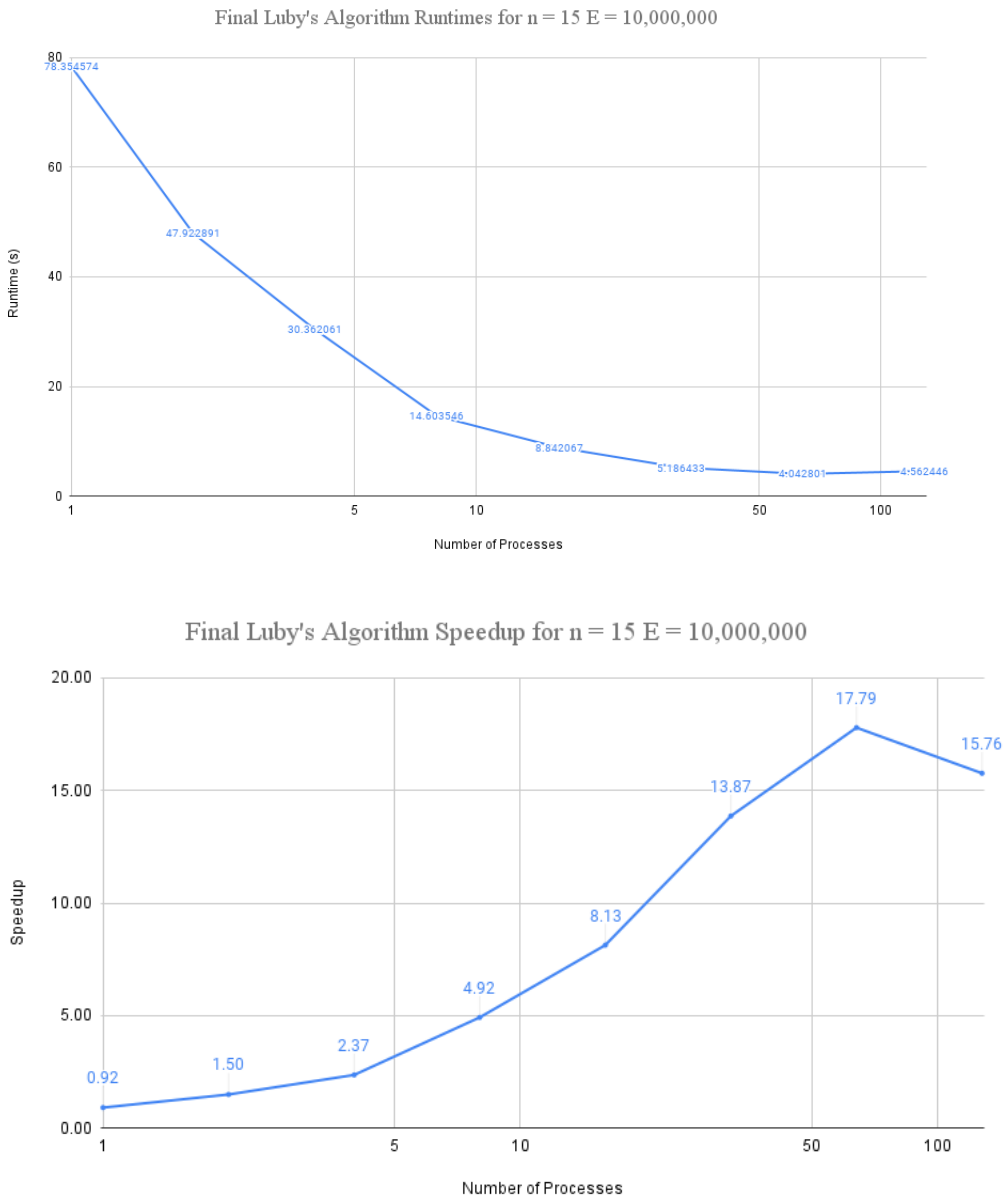


Figure 10: Running time and speedup of V3 on configuration 3, with $1, 2, 4, 8, 16, 32, 64, 128$ processes

Step	Running Time
1	0.008811
2	0.000071
3	0.009873
4	0.000142
5	0.014436
6	0.006535
7	0.003853

Table 2: Running time spent in each step of V3 with 128 processes on configuration 1.

Step	Running Time
1	0.119746
2	0.001874
3	0.021346
4	0.003233
5	0.077348
6	0.013218
7	0.053960

Table 3: Running time spent in each step of V3 with 64 processes on configuration 2.

Finally, we consider the configuration with $10,000,000$ edges. The effect discussed above for configuration 2 is still noticeable in step 7 (see Tables 5 and 6). However, the speedup decreases mainly since the time spent in step 5 greatly increases from 64 to 128 processes, which occurred in configuration 1 as well.

5 Work Done by Each Student

Clarissa

1. Greedy Sequential Algorithm
2. Checker Program for Correctness
3. V1 Luby's Algorithm
4. V3 Luby's Algorithm with Blocked Assignment and Single Messaging
5. Report and Presentation

Arvind

1. Graph Generation algorithm

Step	Running Time
1	0.048356
2	0.000310
3	0.022923
4	0.000484
5	0.021840
6	0.006136
7	0.028327

Table 4: Running time spent in each step of V3 with 128 processes on configuration 2.

Step	Running Time
1	1.905316
2	0.018503
3	0.271207
4	0.035717
5	0.475019
6	0.059983
7	1.251579

Table 5: Running time spent in each step of V3 with 64 processes on configuration 3.

2. Luby's Sequential Algorithm
3. V2 Luby's Algorithm with Blocked Assignment
4. V3 Luby's Algorithm with Blocked Assignment and Single Messaging
5. Report and Presentation

The distribution was 50-50.

References

[BFS12] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. *CoRR*, abs/1202.3205, 2012.

[Ble] Guy E. Blelloch. Introduction to parallel algorithms.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446. SIAM, 2004.

Step	Running Time
1	2.267993
2	0.022872
3	0.262906
4	0.042245
5	1.275207
6	0.047352
7	0.591738

Table 6: Running time spent in each step of V3 with 128 processes on configuration 3.