Luby's Maximal Independent Set Algorithms

Shahin John John Stella
UIN:635008814

May 2025

Abstract

This report investigates the Maximal Independent Set (MIS) problem, with a particular focus on Luby's Monte Carlo Algorithm A and B in a distributed setting using YGM. Both algorithms were implemented, and their performance was analyzed through weak scaling, strong scaling, and degree sensitivity experiments. A key challenge addressed was reducing the number of iterations required to empty the graph, with a targeted bound of O(log n) in expected time. The report presents an intuitive approach to achieving this bound by modifying the random value assignment mechanism for each vertex, thereby exhaustively exploring the choice space to ensure the desired performance. The findings contribute to improving the efficiency of MIS algorithms in parallel and distributed environments.

1 Introduction

A *maximal independent set* (MIS) in an undirected graph is a maximal subset of vertices I such that no two vertices in I are adjacent. The *MIS problem* is to find such a subset. The obvious sequential algorithm for the MIS problem can be simply stated as: Initialize I to the empty set; for $i = 1, \ldots, n$, if vertex i is not adjacent to any vertex in I, then add vertex i to I. However, it is difficult to parallelize this approach. Therefore, we explore the following algorithms.

One notable approach to this problem is **Monte Carlo Algorithm A/B**, which exhibits an appealing structural property: its decomposition into simple, nearly identical sub-algorithms mirrors the natural subdivision present in the input graph. Algorithm A is defined through two algorithmic templates, `ALGVERTEX` and `ALGEDGE`. A separate instance of `ALGVERTEX` is associated with each vertex, and a copy of `ALGEDGE` is assigned to each edge. The algorithm proceeds in multiple phases. In each phase, all instances of `ALGVERTEX` are executed in parallel, followed by a parallel execution of all `ALGEDGE` instances. Remarkably, after only a small number of such phases, the algorithm is likely to produce a maximal independent set. This probabilistic efficiency makes the Monte Carlo approach particularly attractive for distributed systems and parallel protocol design.

To further improve the algorithm, particularly in deterministic settings, we consider **Algorithm C/D**, which employs mutually independent random variables. A key contribution lies in converting this randomized algorithm into a deterministic one, while preserving its running time, but comes at the cost of increased processors count.

1.1 Two Models of P-RAM

Two models of a P-RAM are considered: the CRCW P-RAM, in which concurrent reads and writes to the same memory location are allowed; and the less powerful but perhaps more realistic EREW P-RAM, in which concurrent reads and writes to the same memory location are disallowed. Algorithm A is the simplest Monte Carlo algorithm and has the best running time on a CRCW P-RAM.

Let n be the number of vertices and m be the number of edges in the graph. $EO(k)$ denotes "the expected values is $O(k)$". Luby's algorithms have the following results.

1.2 Motivation

A growing number of parallel algorithms employ the maximal independent set (MIS) algorithm as a subroutine. One notable example is the *maximal matching* problem, which can be efficiently solved

Algorithm	PRAM Type	Processors	Time
A	CRCW	$O(m)$	$EO(\log n)$
B	EREW	$O(m)$	$EO((\log n)^2)$
C	EREW	$O(m)$	$EO((\log n)^2)$
D	EREW	$O(n^2 m)$	$O((\log n)^2)$

Table 1: Time complexity of different algorithms

using MIS. Efficient solutions to this problem enable strong approximation guarantees and effective kernelization strategies, particularly when parameterized by the size of the solution.

For instance, the *Vertex Cover* problem benefits from a kernelization approach based on maximal matching, yielding compact and efficient kernels. Karp and Wigderson [?] demonstrated logspace reductions from the *Maximal Set Packing* and *Maximal Matching* problems to the MIS problem, highlighting the foundational role of MIS in parallel computation.

Independently of Luby's work, Israeli and Itai developed Monte Carlo algorithms for the maximal matching problem. Their algorithm runs in $O(\log n)$ time on a CRCW PRAM and in $O((\log n)^2)$ time on an EREW PRAM.

The reduction of the maximal matching problem to MIS, combined with the results in this paper, establishes a stronger result: there exists a deterministic algorithm for maximal matching that can be implemented on an EREW PRAM with running time $O((\log n)^2)$.

Since I find maximal matching and MIS to be highly useful in the theoretical study of algorithms, I have a natural inclination to explore these algorithms in parallel settings.

2 Monte Carlo MIS algorithms

Below is a general approach for finding a maximal independent set (MIS) in a given graph within a parallel setting. The crux of the algorithm lies in the design of the `select` step, which must satisfy two key properties:

- The `select` step should be implementable on a PRAM such that its execution time is very small.
- The number of iterations of the `while` loop before the graph G' becomes empty should be very small.

The body of the `while` loop, excluding the `select` step, can be implemented on a CRCW PRAM using $O(m)$ processors, where each iteration takes $O(1)$ time. The same portion of the loop can also be implemented on an EREW PRAM using $O(m)$ processors, where each iteration takes $O(\log n)$ time.

Algorithm 1: A high level description of the algorithm

```
  Input: Graph $G = (V, E)$
  Output: A maximal independent set $I$
1 $G' \leftarrow (V', E') \leftarrow (V, E)$;
2 $I \leftarrow \emptyset$;
3 while $G'$ is not empty do
4     Select an independent set $I' \subseteq V'$ in $G'$;
5     $I \leftarrow I \cup I'$;
6     $Y \leftarrow I' \cup N(I')$;
      // $N(I')$ is the set of neighbors of vertices in $I'$
7     $G' \leftarrow$ induced subgraph on $V' \setminus Y$;
8 end
9 return $I$;
```

Figure 1: A high level description of the algorithm

Note: Computing the lexicographically first independent set is P-Complete. (in Step 4)

2.1 Monte Carlo Algorithm A

The `select` step in the above algorithm can be refined using the following implementation.

```
Algorithm 2: Algorithm A: Select Step
1 for $i \in V$ do                                              // ALGVERTEX(i), in parallel
2     $\pi(i) \leftarrow$ a number randomly chosen from $\{1, \ldots, n^4\}$
3 $I \leftarrow V$
4 for $(i, j) \in E$ do                                         // ALGEDGE(i, j), in parallel
5     if $\pi(i) \geq \pi(j)$ then $I \leftarrow I \setminus \{i\}$ ;
6     else $I \leftarrow I \setminus \{j\}$ ;
```

Figure 2: Select Step

At a high level, the goal is to select, in parallel, each vertex that has the lowest priority among its neighbors. The random values $\pi(i)$ chosen in **ALGVERTEX**(i) are mutually independent. The upper bound of n^4 is used to minimize the probability of assigning the same random value to two different vertices. This bound is essentially the square of the number of edges, $|E|^2$. By invoking the birthday paradox, we can show that the probability of a unique assignment is at least $1 - \frac{1}{2n^2}$.

The use of n^4 is important when considering space complexity. However, in my case, I simply use a random generator, which does not have this effect. In fact, it provides a better guarantee due to higher space utilization.

In Luby's algorithm, the input is represented as a list of vertices and a list of edges. Ideally, the availability of shared memory would benefit the algorithm by simplifying communication and coordination between vertices. However, in a distributed setting, such as in YGM, the absence of shared memory necessitates a different approach. The following section describes the implementation details of Luby's algorithm using YGM in a distributed environment.

2.1.1 Data structure

Each vertex in the graph is distributed across processors and maintains the following metadata for use in the Maximal Independent Set (MIS) computation:

- `std::vector<int> edges` – The list of neighboring vertex IDs (i.e., the adjacency list).
- `bool is_removed = false` – Indicates whether the vertex has been removed from the graph during the construction of the MIS. A removed vertex is no longer considered in subsequent iterations.
- `bool is_belong_to_mis = false` – Indicates whether the vertex has been included in the Maximal Independent Set.
- `double random_value = -1.0` – A randomly generated priority value in the interval $(0, 1)$, assigned to the vertex in each iteration. This value is compared with the random values of neighboring vertices to determine MIS candidacy.
- `std::vector<double> neighbor_random_values` – Stores the received random values of all neighboring vertices. Used to determine whether the vertex has the minimum value among its neighbors.

2.1.2 Algorithm Operations

The following is a distributed implementation of Luby's Algorithm for computing a Maximal Independent Set (MIS) using the YGM (Yet-Another-Graph-Mapper) framework. The algorithm proceeds in synchronous rounds with the following major steps:

- **Phase 0: Random Value Assignment and Propagation.** Each active vertex selects a random value in $(0, 1)$ and broadcasts this value to its neighbors. This allows each vertex to compare its value with those of adjacent vertices.

Random value selection: $O(|V|/P)$-time. for all collective is used here.

Random value propagation: $O((|V|+|E|)/P)$-time. For all collective is used to access each vertex and the neighors are iterated over through a for loop.

- **Phase 1: MIS Candidate Selection.** A vertex is added to the MIS if its random value is the minimum among all its neighbors. Once selected, the vertex and all its neighbors are marked for removal to maintain independence.

  Each vertex checks neighbor values: $O((|V| + |E|)/P)$-time. For all collective is used to access each vertex and the neighors are iterated over through a for loop to access their random values for comparison.

- **Phase 2: Removal and Edge Cleanup.** All vertices marked for removal are updated asynchronously. Their outgoing edges are collected and subsequently removed from the graph, ensuring consistency across distributed processes. $O((|V| + |E|)/P)$-time

  Asyncronously visits the collected edges and removes the corresponding edges and vertices from the graph. At most one async visit per removed vertex gives $O(|V|/P)$-time (if MIS selection is sparse), plus removal of edges gives $O(|E|/P)$-time

- **Phase 3: Active Vertex Check.** After each round, a global reduction operation checks whether any active vertices remain. The algorithm proceeds to the next iteration only if there is at least one such vertex.

  All reduce is preformed here with $O(log(P))$-time.

- **Termination and Output.** When no active vertices remain, the algorithm terminates. The root process (rank 0) prints the number of iterations taken, and each process returns the MIS vertices it identified.

Additionally I have barriers in place after each of the above phases, but those account for only $O(|V|/P)$-time

2.1.3 Comparison with the Theoretical result

The paper assumes the presence of $m = |E|$ processors on a CRCW or EREW PRAM model. Under this assumption, each operation inside the `while` loop completes in $O(1)$ time. Additionally, the expected number of iterations is $O(\log n)$. This implies that our implementation in the distributed YGM framework matches the work-efficient model and achieves logarithmic iteration complexity with good scaling, thereby demonstrating efficiency.

The primary difference in my implementation arises from using YGM. In this setting, I must store the random values of all neighboring vertices as part of the adjacency list for each vertex. As the algorithm proceeds, these values need to be updated and retrieved from neighbors, incurring a time cost of $O(d)$, where d is the degree of the vertex. This contrasts with the original version of Luby's algorithm, which reduces this step to $O(\log d)$ time by employing parallel techniques to efficiently find the minimum among neighbor values.

2.2 Monte Carlo Algorithm B

The `select` step in the high level algorithm can be refined using the following implementation to get Monte Carlo Algorithm B. We use the following definition of a random variable for our algorithm, which I refer to as the **Biased Coin Flip Rule**.

For each vertex $i \in V'$, the value of $\text{coin}(i)$ is determined based on its degree $d(i)$ as follows:

- If $d(i) > 0$, then $\text{coin}(i) = 1$ with probability $\frac{1}{2d(i)}$, and $\text{coin}(i) = 0$ otherwise.
- If $d(i) = 0$, then $\text{coin}(i) = 1$ with probability 1.

If the degree of a vertex is zero, then that vertex must be included in the MIS. This selection is guaranteed by assigning $\text{coin}(i) = 1$.

Algorithm 3: Algorithm B: Select Step

```
for $i \in V'$ do                                          // in parallel
    compute $d(i)$
end
$X \leftarrow \emptyset$ ;
for $i \in V'$ do                              // Choice Step, in parallel
    Randomly choose a value for coin($i$) ;
    if $coin(i) = 1$ then  $X \leftarrow X \cup \{i\}$ ;
end
$I' \leftarrow X$ ;
for $(i, j) \in E'$ do                                     // in parallel
    if $i \in X$ and $j \in X$ then
        if $d(i) \leq d(j)$ then  $I' \leftarrow I' \setminus \{i\}$ ;
        else  $I' \leftarrow I' \setminus \{j\}$ ;
    end
end
```

Unlike the previous algorithm, we do not begin with $I = V$ as the initial state. Instead, we use a subset of V, determined by the assigned coin(i) values. This approach helps us capture nuanced vertices that may not have a high degree but still belong in the MIS.

In Luby's algorithm, the input is represented as a list of vertices and a list of edges. Ideally, the availability of shared memory would benefit the algorithm by simplifying communication and coordination between vertices. However, in a distributed setting, such as in YGM, the absence of shared memory necessitates a different approach. The following section describes the implementation details of Luby's algorithm using YGM in a distributed environment.

2.2.1 Data structure

Each vertex in the graph is distributed across processors and maintains the following metadata for use in the Maximal Independent Set (MIS) computation:

- `std::vector<int> edges` – The list of neighboring vertex IDs (i.e., the adjacency list).
- `bool is_removed = false` – Indicates whether the vertex has been removed from the graph during the construction of the MIS. A removed vertex is no longer considered in subsequent iterations.
- `bool is_belong_to_mis = false` – Indicates whether the vertex has been included in the Maximal Independent Set.
- `std::vector<bool> neighbor_coins` – Stores the random coin values received from each neighboring vertex. This is used in randomized comparisons to determine local priority.
- `std::vector<int> neighbor_degrees` – Stores the degrees (i.e., number of neighbors) of adjacent vertices. This information can be used in tie-breaking or in degree-based heuristics during the MIS selection process.

2.2.2 Algorithm Operations

The following is a distributed implementation of Luby's Algorithm B for computing a Maximal Independent Set (MIS) using the YGM (Yet-Another-Graph-Mapper) framework. The algorithm proceeds in synchronous rounds with the following major steps:

- **Phase 0A: Degree Announcement.** Each vertex sends its current degree to all of its neighbors. This information will be used by neighbors in the decision-making process of later phases.

  For all collective is used to acces aach vertex and the neighbors are iterated over through a for loop. This would consume $O((|V| + |E|)/P)$-time.

- **Phase 0B: Tentative MIS Selection via Coin Flip.** Each vertex that has not yet been removed performs a probabilistic coin flip to decide whether to tentatively join the MIS:

  - If $d(i) > 0$, then $\text{coin}(i) = 1$ with probability $\frac{1}{2d(i)}$, and $\text{coin}(i) = 0$ otherwise.
  - If $d(i) = 0$, then $\text{coin}(i) = 1$ with probability 1.

  Vertices with $\text{coin}(i) = 1$ are marked as tentative MIS candidates.

  coin value assigngment is done through a fro all collective and takes $(O(|V|/P))$-time.

- **Phase 0C: Coin State Propagation.** Each vertex communicates its coin value (i.e., tentative MIS decision) to all of its neighbors. This allows each vertex to know whether any adjacent vertex has also decided to tentatively join the MIS.

  For all collective is used to update the neighbors coin values. This would take $O((|V| + |E|)/P)$-time.

- **Phase 1: Finalization and Conflict Resolution.** Each vertex that has tentatively joined the MIS checks its neighbors:

  Each vertex checks neighbor values: $O((|V| + |E|)/P)$-time. For all collective is used to access each vertex and the neighors are iterated over through a for loop to access their random values for comparison.

  - If a neighbor also tentatively joined and has a higher degree, the vertex withdraws from the MIS.
  - If a neighbor has the same degree and a lower vertex ID, the vertex also withdraws.

  If the vertex remains a valid candidate after this check, it is added to the MIS. The vertex and all its neighbors are then marked as removed from the graph for future iterations.

- **Phase 2: Removal and Edge Cleanup.** All vertices marked for removal are updated asynchronously. Their outgoing edges are collected and subsequently removed from the graph, ensuring consistency across distributed processes. $O((|V| + |E|)/P)$-time

  Asyncronously visits the collected edges and removes the corresponding edges and vertices from the graph. At most one async visit per removed vertex gives $O(|V|/P)$-time (if MIS selection is sparse), plus removal of edges gives $O(|E|/P)$-time

- **Phase 3: Active Vertex Check.** After each round, a global reduction operation checks whether any active vertices remain. The algorithm proceeds to the next iteration only if there is at least one such vertex.

  All reduce is preformed here with $O(log(P))$-time.

- **Termination and Output.** When no active vertices remain, the algorithm terminates. The root process (rank 0) prints the number of iterations taken, and each process returns the MIS vertices it identified.

2.2.3 Comparison with the Theoretical result

The primary difference in my implementation arises from using YGM. In this setting, I must store the random coin values of all neighboring vertices, as well as the degree (updated degree counts) as part of the adjacency list for each vertex. As the algorithm proceeds, these values need to be updated and retrieved from neighbors, incurring a time cost of $O(d)$, where d is the degree of the vertex. This contrasts with the original version of Luby's algorithm, which reduces this step to $O(\log d)$ time by employing parallel techniques to efficiently find the minimum among neighbor values.

The following is a summary of the comparison with the theoretical results for both Algorithm A and Algorithm B.

	Overall Runtime	**Algorithm-A (Least Priority)**	**Algorithm-B (Least Degree)**
Theoretical Result	$O(\log \lvert V\rvert \cdot \log d)$ (w.h.p.)	$O(\log d)$	$O(\log d)$
My Implementation	$O(\log \lvert V\rvert \cdot d)$ (w.h.p.)	$O(d)$	$O(d)$

Table 2: Comparison of Theoretical vs. Implemented Runtime Complexities

3 Experimental Studies

3.1 Weak Scaling Results

For this test, the number of vertices ranges from 1,250 to 10,000, the number of edges ranges from 60,000 to 480,000, and the processor count ranges from 1 to 48.

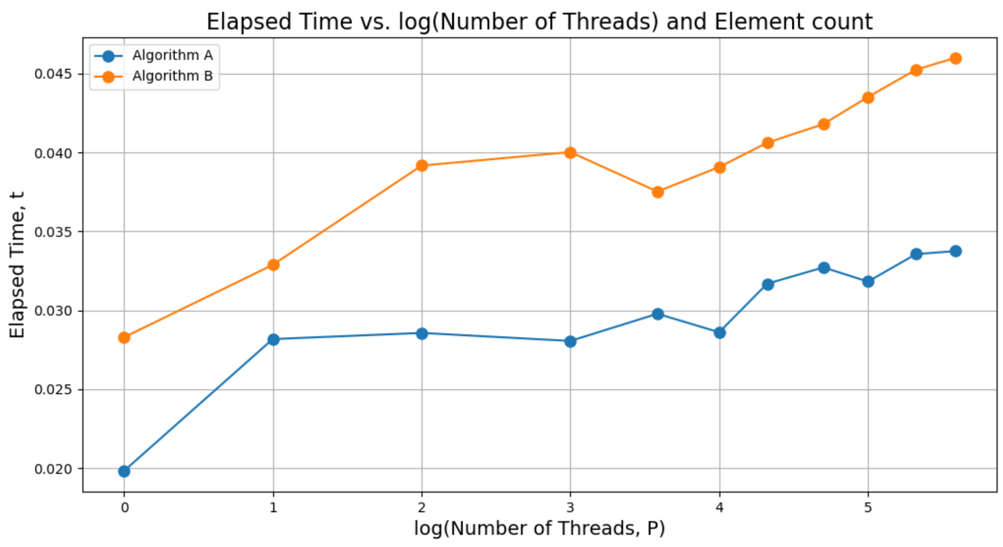

Figure 3: Weak Scaling Results

The time intervals between execution measurements are small, with minor fluctuations observed in the plot. These fluctuations can be partly attributed to the varying number of iterations in the while loop and partly to the graph's overall structure and degree variance. Overall, the plot remains nearly horizontal with respect to the x-axis, which is characteristic of good weak scaling performance.

3.2 Strong Scaling Results

For this test, the number of vertices is 80,000, the number of edges is 640,000, and the processor count ranges from 1 to 48.

As the number of processors increases, we observe a decrease in the elapsed time. This behavior aligns with the goals of strong scaling and effectively demonstrates the algorithm's strong scaling capability.

3.3 Degree Experiment: Time vs. Average Degree

For this test, the number of vertices is 40,000, the number of edges ranges from 40,000 to 2,560,000, and the processor count is 32.

Here, the goal was to understand how the average degree affects the time complexity. In **Algorithm A**, we need to compare a vertex's d neighbors to identify the one with the lowest priority (random) value. Only after this comparison can we determine whether the vertex itself should be included in the independent set. In **Algorithm B**, we again compare d neighbors, but this time to find the one with the lowest degree among the neighbors of a vertex in the independent set. Therefore, as the average

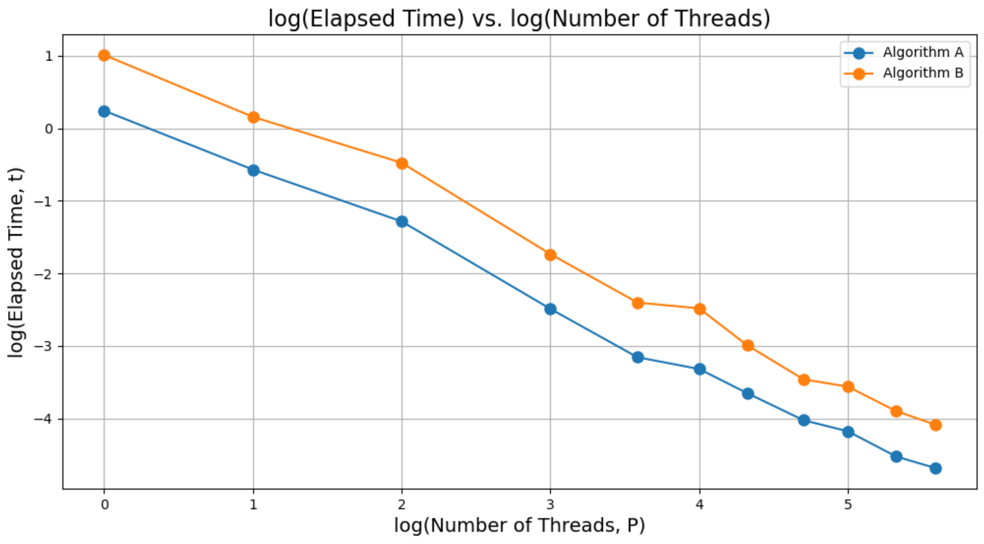

Figure 4: Strong Scaling Results

degree increases, the time complexity also increases due to the growing number of comparisons required for each vertex.

According to the referenced paper, this comparison step involves a $\log n$-time reduction to find the minimum priority value in Algorithm A, and similarly, the minimum degree value in Algorithm B. However, in our implementation using **YGM**, since vertices are stored using an adjacency list along with associated metadata (neighbor priority/coin state and degree count), the reduction—combined with asynchronous `visits` to update neighbor metadata—takes $O(d)$ time in the distributed setting. This explains the observed increase in elapsed time as the average degree of the graph increases.

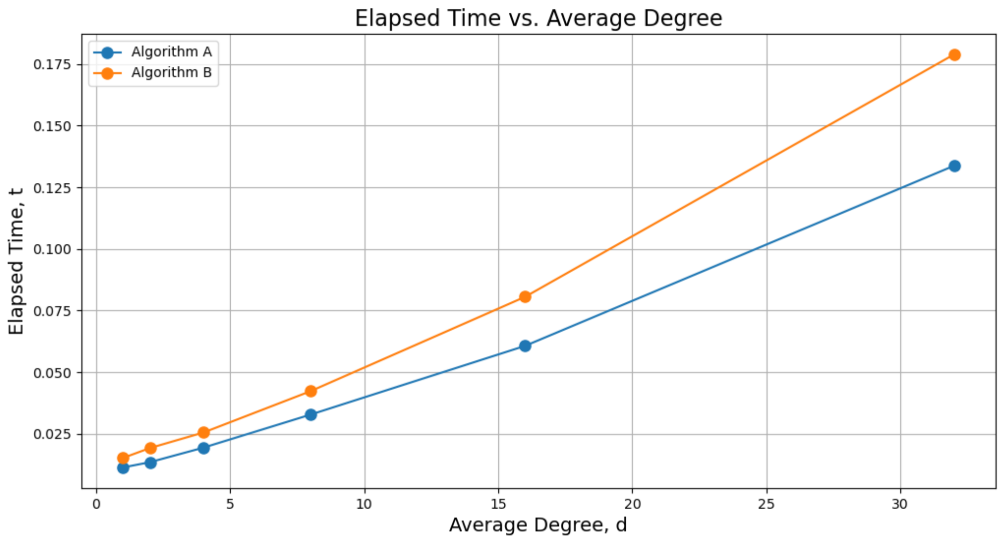

Figure 5: Degree Study Results

3.4 Degree Experiment: Number of Iteration vs. Average Degree

For this test, the number of vertices is 40,000, the number of edges ranges from 40,000 to 2,560,000, and the processor count is 32.

Based on the above observation, we tried to understand the implication this degree increase had on the number of iterations of the algorithm. It was observed that with an increase in degree count, there was a slight increase in the number of iterations. There are fluctuations due to the graph structure and the variation in the degree spread across the graph. However, there is still the cost of communicating and fetching the minimum degree values from both algorithms.

In **Algorithm A**, we are at the phase where we are trying to reduce conflicts. For instance, consider an edge where both vertices initially belong to the independent set (i.e., $I = V$). If both vertices have degree 1, then the removal of one vertex is only dependent on its neighboring vertex. As the degree increases, there are more vertices to consider. Consequently, the chance of both vertices being removed from the graph becomes higher, because the likelihood of either vertex having the lowest priority random value among its neighbors diminishes. Although the number of iterations increases only by 1 or 2, this is actually significant since we know the algorithm terminates in $O(\log V)$ steps. Since we are counting iterations in logarithmic terms, this small increase becomes meaningful.

The same argument applies to **Algorithm B**, with the distinction being that the decision is based on the degree value instead of the priority random value. The small fluctuation in the number of iterations here can also be attributed to the fact that we start with a coin-flip-based binary probability for each vertex's inclusion in the set X. This means that unlike Algorithm A, which starts from the full vertex set V, Algorithm B starts from a subset of V constructed using the coin-flip probability. This distinction explains the increased number of iterations in Algorithm B compared to Algorithm A. However, the increase is not very significant, it is only short by a constant small factor, as evident from the plot.

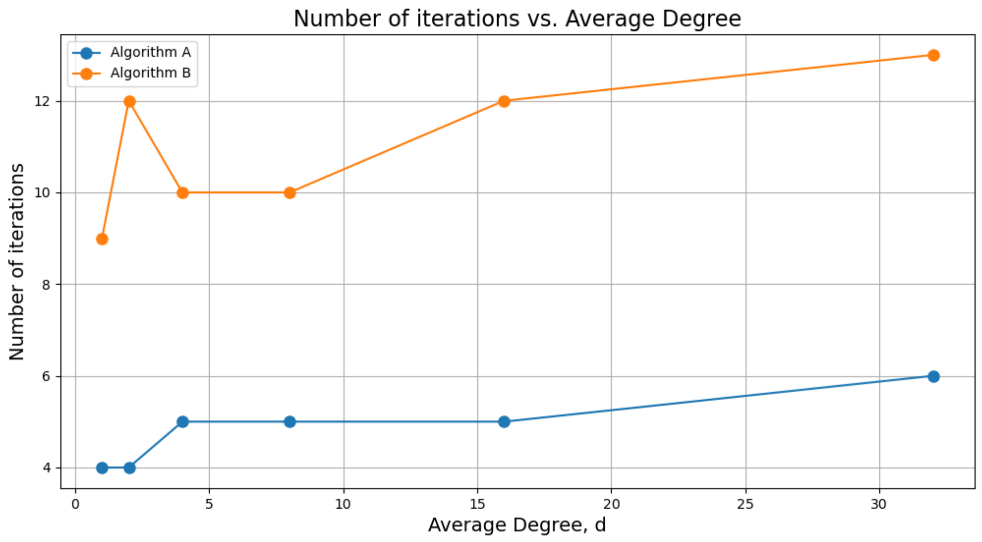

Figure 6: Degree Study Loop Count Results

4 Algorithm C and D: Intution behind it

To convert a parallel Monte Carlo algorithm into a deterministic one, we focus on its randomized choice step, where independent random variables are selected. If a good assignment of these variables leads to correct results and can be efficiently verified, then the algorithm can be de-randomized by systematically searching for such good assignments.

To convert a parallel Monte Carlo algorithm B into a deterministic one, we focus on its randomized choice step, where independent random variables are selected. A good assignment of these variables leads to correct results and can be efficiently verified. The deterministic version mirrors the Monte Carlo algorithm, but instead of relying on randomness, we exhaustively evaluate a small, representative set of possible assignments in parallel. This is done by constructing a finite probability space with a small number of samples, each representing a possible assignment. Since the analysis guarantees that a good assignment exists with positive probability, running multiple instances in parallel ensures that at least one will be good. Once a valid assignment is found, the algorithm proceeds as in the original, turning the expected $EO(\log n)$ runtime into a worst-case $O(\log n)$ runtime.

5 Conclusion

This study was highly beneficial in advancing my understanding of the Maximal Independent Set (MIS) problem, particularly in the context of Luby's Monte Carlo Algorithm A and B within a distributed setting using YGM. Through experimenting with weak and strong scaling, as well as analyzing degree sensitivity, I was able to successfully reduce the number of iterations needed to empty the graph, achieving the targeted O(log n) expected time bound. The modification of random value assignments for each vertex was a key insight in exhaustively exploring the choice space, ultimately optimizing the algorithm's performance. This work has deepened my knowledge of efficient MIS computation and provided valuable lessons for optimizing parallel algorithms in large-scale systems.