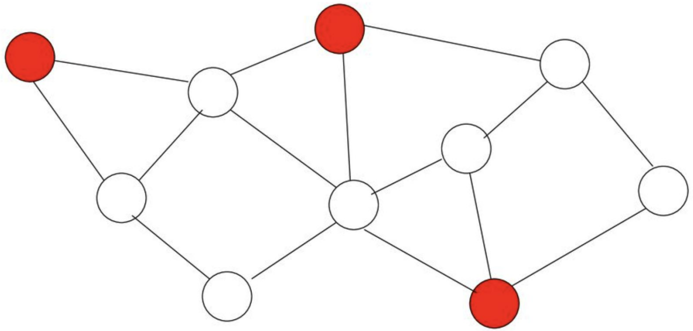
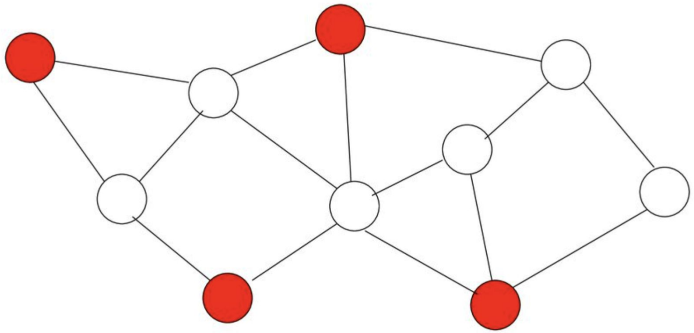
Luby’s Algorithm for Maximal Independent Set

What is Maximal Independent Set (MIS)?

Independent Set (IS): Any set of nodes that are not adjacent



Maximal Independent Set: An independent set that is no subset of any other independent set



A General Algorithm for Computing MIS

Algorithm 1: A high level description of the algorithm

Input: Graph $G = (V, E)$

Output: A maximal independent set I

$G' \leftarrow (V', E') \leftarrow (V, E)$;

$I \leftarrow \emptyset$;

while G' *is not empty* **do**

    Select an independent set $I' \subseteq V'$ in G';

    $I \leftarrow I \cup I'$;

    $Y \leftarrow I' \cup N(I')$;

    // $N(I')$ is the set of neighbors of vertices in I'

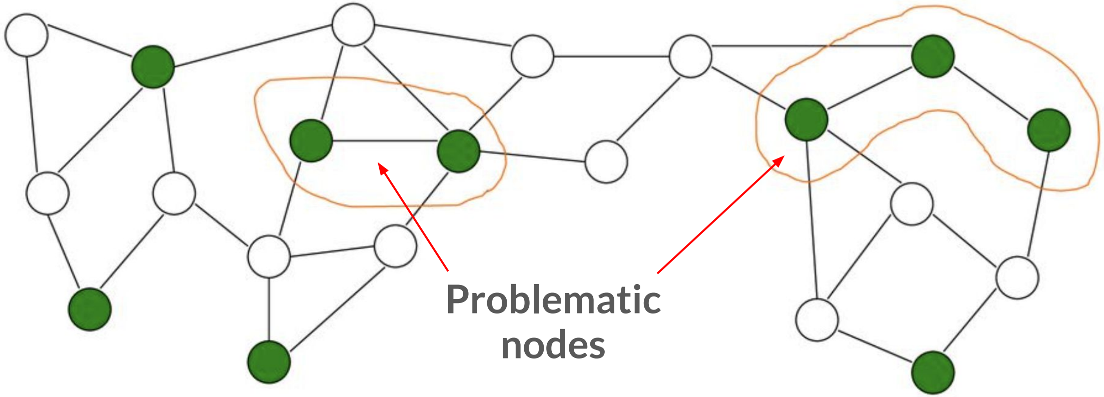
    $G' \leftarrow$ induced subgraph on $V' \setminus Y$;

end

return I;

Observation: The Select Step

- The **number of iterations** depends on the **choice of independent set** in each iteration.
- The **larger the independent set** at each iteration the **faster the algorithm**.



Monte Carlo Algorithm A

Algorithm 2: Select Step

for $i \in V$ **do** `// ALGVERTEX(i), in parallel`

  $\pi(i) \leftarrow$ a number randomly chosen from $\{1, \ldots, n^4\}$

$I \leftarrow V$

for $(i, j) \in E$ **do** `// ALGEDGE(i, j), in parallel`

  if $\pi(i) \geq \pi(j)$ **then**

    $I \leftarrow I \setminus \{i\}$

  else

    $I \leftarrow I \setminus \{j\}$

Monte Carlo Algorithm B

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

Monte Carlo Algorithm B

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

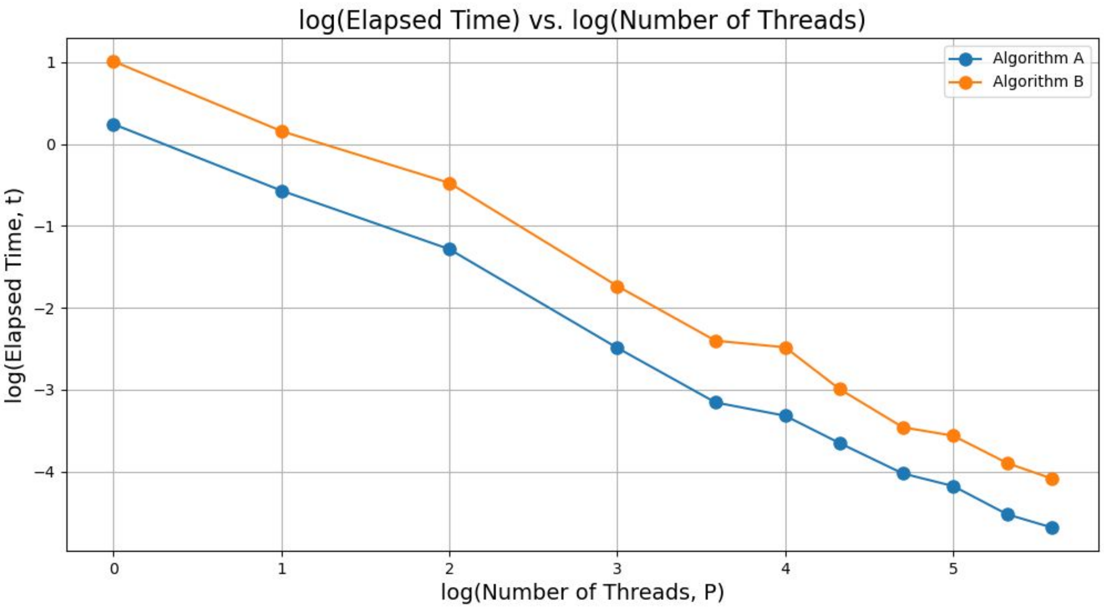
Distributed Algorithm with YGM

	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

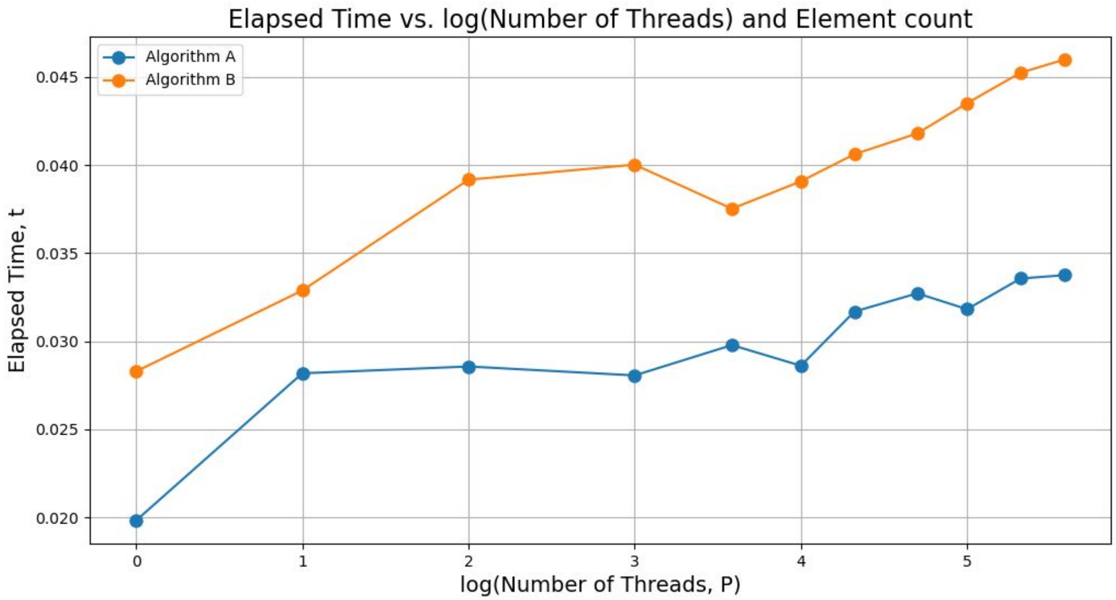
- Theoretical Result: O(log d)-time reduction to find the neighbour with the least priority (Algorithm A) / degree (Algorithm B).
- My Implementation: O(d)-time to find the neighbour with the least priority (Algorithm A) / degree (Algorithm B).

Strong Scaling Results



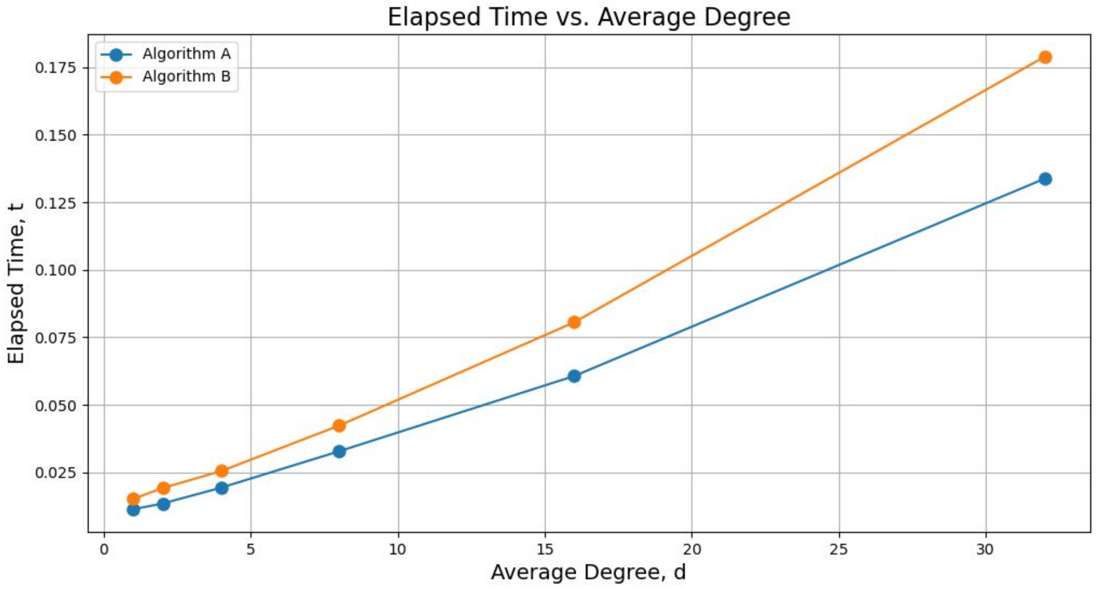
Number of Vertices: 80,000 | Number of Edges: 640,000 | Processor Count: 1 to 48

Weak Scaling Results



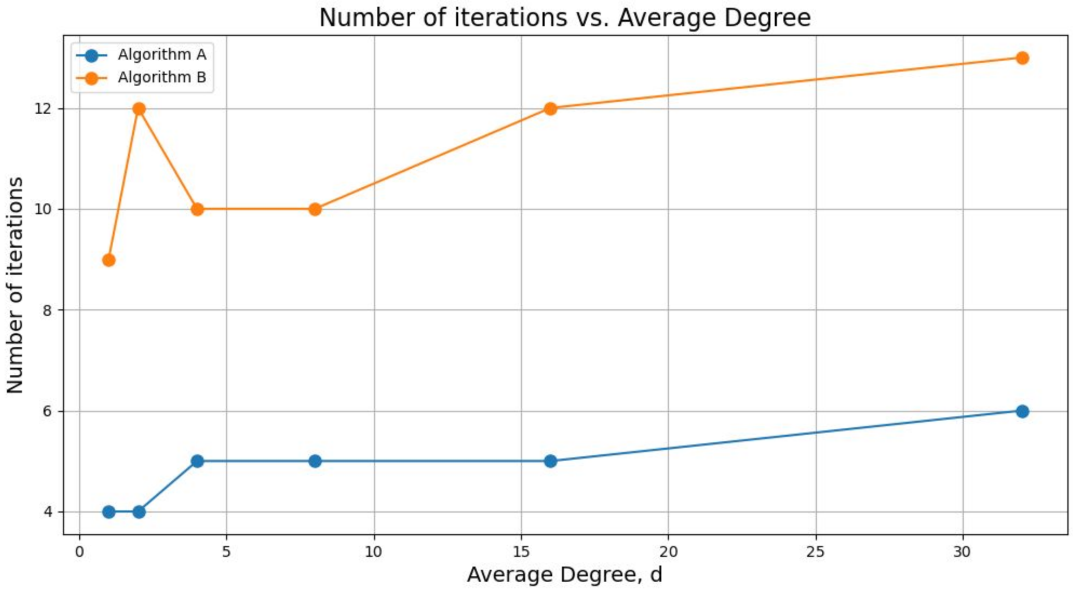
Number of Vertices: 1,250 to 10,000 | Number of Edges: 60,000 to 480,000 | Processor Count: 1 to 48

Degree Experiment - Time vs Degree



Number of Vertices: 40,000 | Number of Edges: 40,000 to 2,560,000 | Processor Count: 32

Degree Experiment - Iterations vs Degree



Number of Vertices: 40,000 | Number of Edges: 40,000 to 2,560,000 | Processor Count: 32

Removing randomization from parallel Algorithms

De-randomization strategy: Monte Carlo -> Deterministic Algorithm

1. **Isolate a random choice step** in the algorithm
2. Construct **a small set of independent random variables**
3. Run the algorithm over **all choices from this set.**

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	$O(n^2 m)$	$O((\log n)^2)$

This is very powerful in parallel setting:

- All candidate choices can be evaluated from a bounded space simultaneously across processors -> guarantee success