Parallel algorithms for the maximal independent set problem in graphs

Luis Barba
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
luis_barbaflores@carleton.ca

December 10, 2012

Abstract

In this paper we describe the randomized parallel algorithm proposed by Blelloch et al. [BFS12] to compute a Maximal Independent Set (MIS) of a given graph. We implemented their algorithm as well as the fastest sequential algorithm and compared their performance with different input graphs. Furthermore, we compared the number of rounds in both the sequential and parallel algorithms and present empirical evidence of the intrinsic parallelism in the algorithm proposed by Blelloch et al. [BFS12].

1 Introduction

Let $G = (V, E)$ be an undirected graph. An *independent set* of G is a subset I of V such that no two vertex in I are adjacent in G. A *Maximal Independent Set* (MIS) of G is an independent set such that adding any other vertex to it forces the set to contain an edge between two of its vertices. A Maximum independent Set (MaxIS) is a largest maximal independent set contained in V; see Fig. 1. The problem of finding a MaxIS has been proved to be NP-hard. Moreover, it is even hard to find an ε-approximation unless $P = NP$ [EH00]. However, finding a MIS is relatively easy and many (greedy) linear time algorithms are known to solve this problem. The idea to construct a MIS I is usually the following: Given an ordering of the vertices of G, process each vertex by adding it to I as long as it is not adjacent to a vertex already in I. In this way, it is not hard to see that a MIS would be produced. The output of this algorithm is known as the Lexicographical Maximal Independent Set (LFMIS) for the given order. In fact, if the vertices are sorted by degree, from minimum to maximum, then the output of the algorithm is a $\frac{1}{\Delta+1}$-approximation of the MaxIS, where Δ is the maximum degree of a vertex in G. That is, the size of the computed approximation MIS has at least $\frac{k}{\Delta+1}$ where k is the size of a MaxIS in G.

In this work we are interested in the study of the MIS and LFMIS problem in parallel computing. In this setting, the MIS problem has been extensively studied: Vliant [Val82] noted that the MIS problem, which has an easy sequential algorithm, may be one of the problems for which no fast parallel algorithm exists. Cook [Coo85] strengthened this belief by proving that obtaining the LFMIS for any arbitrary ordering of V is P-complete. Where P is the class of all problems solvable in polynomial time. In other words, there is no

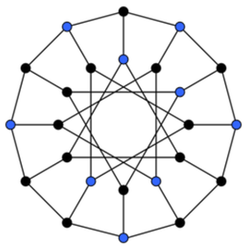


Figure 1: A Maximum Independent Set of the graph is depicted in blue.

parallel algorithm to find the LFMIS for any given order of V unless $P = NC$. Therefore, to solve the MIS problem, researches needed to look for a different approach.

The study of the MIS problem in the parallel setting was inspired by the growing number of parallel algorithms using the MIS algorithm as a subroutine. Karp and Wigderron [KW85] gave NC^1 reductions from the Maximum Set Packing and the Maximal Matching problems to the MIS problem, and an NC^2 reduction from the 2-Satisfiability problem to the MIS problem. Luby [Lub86] also showed an NC^1 reduction from the Maximal Coloring problem to the MIS problem.

Karp and Wigderson [KW85] surprised the research community by developing a fast parallel algorithm for the MIS problem. They presented a randomized algorithm with expected running time $O(\log^4 n)$ using $O(n^2)$ processors, and a deterministic algorithm with running time $O(\log^4 n)$ using $O(\frac{n^3}{\log^3 n})$ processors on a EREW PRAM. That is, they established that the MIS problem is in NC^4. Independently, Goldberg and Spencer showed a deterministic parallel algorithm running in $O(\log^4 n)$ time on the EREW PRAM model [GS89]. Later on, Luby [Lub86] and Alon et al. [ABI86], independently proved the existence of a Monte Carlo algorithm in the EREW PRAM running in expected $O(\log^2 n)$ time using $O(|E|)$ processors. Furthermore, Alon et al. showed that in the CRCW PRAM, the MIS problem can be solved in $O(\log n)$ time using $O(|E|\Delta)$ processors. Both papers show a way to derandomized their algorithms at the expense of using $O(mn^2)$ processors, obtaining in this way efficient deterministic parallel algorithms for the MIS problem.

A few years later, Coppersmith [CRT89] studied the problem in the setting of random graphs. He showed that in a random graph with n vertices, the LFMIS can be found in $O(\log^{O(1)} n)$ expected time using linearly many processors in the CRCW PRAM model for any given order. Recall that LFMIS is P-complete for general graphs. However, this is not contradiction and simply shows that in the "average" case, the LFMIS problem is highly parallelizable. Calkin and Frieze further analyzed this algorithm and proved that the expected running time of Coppersmith's algorithm is in fact $O(\log n)$ for random graphs of arbitrary edge density. Nevertheless, these results were only theoretical and couldn't be implemented to solve real-life applications.

The greedy sequential algorithm to compute a MIS loops over the vertices according to the given order, adding them to the resulting set only if no previous neighboring vertex has been added. In this loop, each iterate depends only on a subset of the previous iterates. That is, to decide the fate of a vertex we need only to know if any one of the vertex's previous neighbors (in the given order) is in the partially computed MIS. This leads to a dependence structure among the iterates. If this structure is shallow (has a polylogarithmic depth), then running each of the iterates in parallel, while respecting the dependencies, leads to an efficient parallel implementation that mimics the sequential algorithm. The depth of this dependance structure is called the *dependence length*.

Using these abstraction, Blelloch et al. [BFS12] revisited the ideas of Coppersmith and Calkin et al., and showed that by randomly permuting the order in which the vertices are given, the dependence length of the sequential algorithm for the LFMIS problem has $O(\log^2 n)$ depth with high probability, where the probability is with respect to the random permutation. That is, for almost every permutation of the vertices the dependance structure will be shallow. Furthermore, this result was followed by a new parallel implementation providing a continuous trade-off between total work and running time. To achieve this result, they fix a permutation of the vertices and instead of processing all of them at once, they process only a prefix of the vertices in parallel. By reducing the size of the prefix, the parallelism is reduced but so is the redundant work. When the prefix is of size one, the algorithm mimics the sequential algorithm with no redundant work. Using this trade-off, they showed that by choosing and appropriate size for the prefix, they can obtain linear work while computing the solution after a polylogarithmic number of steps. Another interesting property of their algorithm is that even though their algorithm is randomized, it produces the same output whenever the same ordering of the vertices is given as input. This can be a desirable property for parallel algorithms as shown in [BAAS09].

Results

We implemented both the parallel algorithm presented by Blelloch et al. [BFS12] and the sequential algorithm in Cilk and C, respectively. The experiments we ran provide empirical proof of the shallowness of the dependence structure of the parallel algorithm. That is, we prove that the theoretical results that Blelloch et al. [BFS12] presented hold for several benchmark graphs. Furthermore, we provide empirical evidence of the advantage of using a parallel approach to compute a MIS.

Our benchmarks are based on sparse graphs such as the 3D-grid, hypercube and random graphs, where the probability of the existence of an edge is inversely proportional to the distance between the indices of the vertices. These graphs have been the benchmarks used to test similar algorithms [CZF04, BPT11, BFS12].

In Section 3 we describe the classical sequential algorithm to compute a MIS, as well as the parallel algorithm presented by Blelloch et al. [BFS12]. We give an overview of its properties and provide the intuition behind its efficiency. In Section 4 we show the results we obtained through our implementation and describe the empirical evidence that they provide.

Related work and applications

Given a graph G, the *Set Cover* problem asks for the smallest set of points (called a vertex cover) such that every edge of the graph is incident to an element of the vertex cover. The relation between Set Cover and MIS is tight since a set of vertices is a vertex cover if and only if its complement is an independent set. An immediate consequence is that a big MIS produces a small vertex cover.

While a vertex cover can be found in polylogarithmic time using parallel algorithms for MIS, its output may be arbitrarily big (bad) compared with the size of a minimum vertex cover. Therefore, another branch of study is dedicated to compute good approximations for graph problems using parallel algorithms for MIS.

While the Set Cover problem is NP-hard, good approximation algorithms exist in the sequential setting as shown by Chvatal in [Chv79]. Blelloch et Al. [BPT11] studied the Set Cover problem in the parallel setting and showed the existence of a $(1+\varepsilon)H_n$-approximation parallel algorithm using linear work, where $H_n = \sum_{k=1}^{n} \frac{1}{k}$. To obtain this result they use a set of tools related to MIS's. They introduced the concept of Maximal Nearly Independent Set (MaNIS) and presented an $O(m)$ work and $O(\log^2 m)$-time parallel algorithm for the MaNIS problem in the EREW PRAM model. The MaNIS problem is to find a subset of the power set of the vertices of G such that they are nearly independent (their elements do not overlap too much), and maximal (no set can be added without introducing too much overlap). The MaNIS abstraction generalize de MIS problem and allows them to use the duality to solve several set-cover like problems.

The MIS problem is a basic problem in Graph Theory since several problems can be reduced to it. For example, finding small proper colorings or finding maximal matchings in a graph. A *vertex coloring* of a graph $G = (V, E)$ is an assignment of colors to the vertices of G. A vertex coloring is *proper* if no two adjacent vertices are assign with the same color. A proper coloring of G is minimum if G cannot be properly colored with less colors. One important remark is that every chromatic class of a proper coloring is a MIS of G. Therefore, one way to compute a proper coloring with "few" colors is to compute a large MIS, assign the same color to all its vertices and remove it from G afterwards. By repeating this process recursively, we obtain an algorithm to compute a proper coloring of G. However, the quality of the coloring depends heavily on the choice for the MIS algorithm. Therefore, having good and fast algorithms to compute MIS translates into efficient algorithms to compute a proper coloring of G.

Another classical problem in graph theory is the matching problem. A set of edges of a graph G is a *matching* if no two edges in it share an endpoint. The problem of computing a Maximal Matching (MM) can also be solved using an algorithm for MIS. Consider the edge-graph of G. That is, the graph whose vertex set are the edges of G and where two of these edges are adjacent if they share a common endpoint. Therefore, a MIS of the edge-graph of G corresponds to a MM of G. While this algorithm is straightforward, it requires the computation of the edge-graph of G, which can be expensive. Therefore, direct parallel algorithms are of great interest. In this direction, Blelloch et al. in [BFS12] provided a variation of their techniques to solve the MIS and applied them directly to find a MM.

Other problems such as the vertex covering and maximal clique problems are also closely related to MIS. The consequences and applications of a good algorithm to find a MIS are too many and just a few are mentioned in this work. These remarks, together with the fact that finding a MaxIS is NP-hard and a LFMIS is P-complete, show that the study of this problem is fundamental and that the consequences of any development in this field are of great importance.

2 Preliminaries

Let $G = (V, E)$ be an undirected graph such that $|V| = n$ and $|E| = m$. Given two vertices $v, u \in V$, we say the u and v are *neighbors* if the edge $uv \in E$. For each vertex $v \in V$, let $N(v) = \{u \in V : uv \in E\}$ be the *neighborhood* of v. Let $\delta(v) = |N(v)|$ be the *degree* of v and let Δ be the maximum degree among the vertices in V. Given a subset $W \subset V$, let $N(W) = \cup_{w \in W} N(w)$ be the *neighborhood* of the set W. Given a set $V' \subseteq V$, let $G[V'] = (V', E')$ be the *induced* graph by V' where $E' = \{uv : u, v \in V'\}$.

Let π be a total ordering of the vertices of G. Given two vertices $v, u \in V$, we say that

u has *higher priority* that v if u lies before v according to π.

A directed graph is a graph where each edge has an orientation from one endpoint to the other. Given a directed graph, the in-degree of a vertex v is the number of edges incident on v that are directed towards v.

3 The algorithm

In this section we provide a quick overview of the algorithm presented in [BFS12] and the techniques used to prove its efficiency. To do this, we first revisit the classical sequential algorithm to compute a MIS and show how to modify it to work in the parallel setting. All algorithms presented in this section are assumed to be in CRCW PRAM model.

The sequential algorithm for the MIS problem is simple: Take the vertex v with highest priority, according to π, among the remaining vertices in the graph, add v to the MIS and then remove v and all vertices in $N(v)$ from the graph. Repeat this process until the graph is empty. The pseudo code of this algorithm is shown in Algorithm 1. The MIS returned by this algorithm is called the *LFMIS with respect to the ordering* π or π-MIS for short.

Algorithm 1 Sequential greedy algorithm for the MIS problem

```
1: procedure SequentialGreedyMIS($G, \pi$)
2: if $|V| = 0$ then
3:     return $\emptyset$
4: else
5:     let $v$ be the first vertex in $V$ according to $\pi$
6:     $V' = V \setminus (v \cup N(v))$
7:     return $v \cup$ SequentialGreedyMIS($G[V'], \pi$)
8: end if
```

The idea of the parallel algorithm will be to process many vertices as in Algorithm 1 at the same time. However, we have to be careful as we want to avoid a situation in which a vertex u is added to the MIS by one process while discarded by some other process running in parallel.

Notice that if a vertex v has no higher-priority neighbors in the graph, then it will be added to the MIS regardless of what happens with vertices of higher priority than v. This is the key fact to obtain parallelism from Algorithm 1, i.e., if we process all vertices with no higher-priority neighbors in the graph at the same time, they won't interfere with each other. By allowing the vertices to be added to the MIS as long as they don't have higher-priority neighbors, we obtain Algorithm 2.

Algorithm 2 Parallel greedy algorithm for the MIS problem

```
1: procedure ParallelGreedyMIS($G, \pi$)
2: if $|V| = 0$ then
3:     return $\emptyset$
4: else
5:     let $W$ be the set of vertices with no higher-priority neighbors (based on $\pi$)
6:     $V' = V \setminus (W \cup N(W))$
7:     return $W \cup$ ParallelGreedyMIS($G[V'], \pi$)
8: end if
```

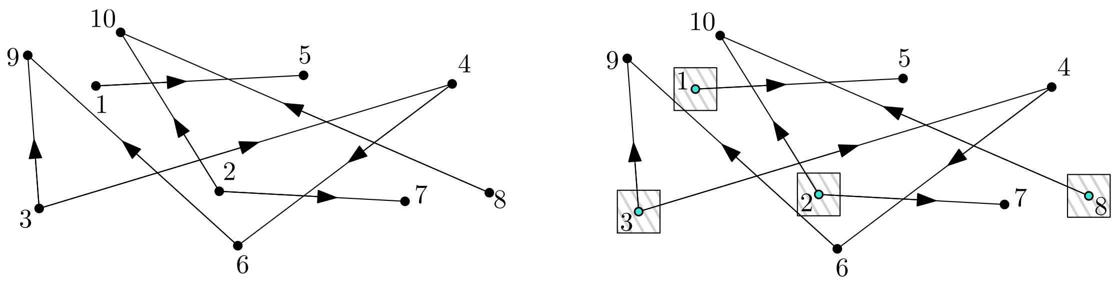


Figure 2: Left: The priority DAG after directing the edges from of higher to the lower priority endpoint. Right: The roots of the priority DAG for the given ordering of the vertices.

Lemma 1 *Given a total ordering π of the vertices of G. Parallel Algorithm 2 returns the same π-MIS as sequential Algorithm 1.*

Even though both algorithms return the same π-MIS, in order to have any speedup in Algorithm 2, we need to accept vertices into the MIS at earlier times than the sequential algorithm. In other words, we need to have many vertices with no higher-priority neighbors on each recursive call of the algorithm that can be processed in parallel.

Notice that if Algorithm 2 computes a new random permutation π on each recursive call, we effectively have the same algorithm presented by Luby [Lub86]. Thus, the key fact is the use of a single permutation through the whole process, this makes the analysis of the algorithm more difficult but provides better results.

Priority Directed Acyclic Graph

The structure of G, together with the given ordering π, provide a dependence structure on the vertices of G that can be better understood in terms of a directed acyclic graph (DAG). The *priority DAG* of G with respect to π is obtained by directing the edges of G from higher priority to lower priority endpoints (according to π). A *root* in the priority DAG is a vertex with in-degree zero; see Fig. 2.

Each recursive call of Algorithm 2 is called a *step.* Notice that on each step, Algorithm 2 process all the roots of the priority DAG in parallel and removes them along with their children from the priority DAG. The number of steps that Algorithm 2 requires before obtaining an empty priority DAG is called the *dependence length.* Though the length of the longest directed path in the priority DAG upper bounds the dependence length, they can be quite different as it is the case in a tournament (complete directed graph). Thus, a different technique needs to be used to bound the dependence length.

The idea is to take a random permutation π and show that with high probability, the dependence length of the priority DAG based on π is polylogarithmic. Blelloch et al. [BFS12] showed this to be true. They considered the priority sub-DAG's induced by small subsets of vertices and proved that they have small longest paths and hence small dependence length (although this may not be true for the complete DAG). Aggregating all these bounds on the sub-DAG's provides an upper bound on the total dependence length.

Specifically, they proposed a more restricted, less parallel algorithm whose pseudo-code can be found in Algorithm 3. This algorithm considers only a subset of the remaining vertices in the graph rather than all of them on each of its recursive calls. This may cause some vertices to be processed later than they would be in Algorithm 2; however, no vertex

Algorithm 3 Prefix parallel greedy algorithm for the MIS problem

```
procedure PrefixParallelMIS($G = (V, E), \pi$)
if $|V| = 0$ then
    return $\emptyset$
else
    Let $P_\delta$ be the subset containing the first $\delta$ vertices of $V$ according $\pi$.
    let $W$ = ParallelGreedyMIS($G[P_\delta], \pi$)
    $V' = V \setminus (P_\delta \cup N(W))$
    return $W \cup$ PrefixParallelMIS($G[V'], \pi$)
end if
```

is processed before. Therefore, when we add up all steps performed in the recursive calls to ParallelGreedyMIS, Algorithm 3 requires more steps than Algorithm 2. Consequently, by showing a polylogarithmic bound on the number of steps of Algorithm 3, we implicitly obtain an upper bound for the number of steps in Algorithm 2.

Let π be a random ordering of the vertices in V. Each recursive call of Algorithm 3 is called a *round*. For any parameter δ a δ-prefix is the subset containing the first δ vertices of V according to the ordering π. While considering larger prefixes of vertices on each round decreases de numbers of rounds, it also increases the number of steps in each round. With some fine tuning of the size of the subset we obtain a polylogarithmic bound on both, the number of rounds and the steps in each round.

The main advantage in the analysis of Algorithm 3 is that the order of the vertices left in the graph remains uniform. Due to lack of space, we will not go into details of the proof but mention the highlights and the main ideas presented by Blelloch et al. [BFS12].

The first step in the proof is to show that after the i-th round, the degree of each vertex in the graph is at most $\Delta/2^i$ with high probability. Hence, after $\log \Delta$ rounds, all vertices have degree 0 with high probability and can be processed in one more round.

Theorem 1 *(Rephrasing of Corollary 3.2 of [BFS12]) By setting $\delta = \Omega(\frac{2^i \log n}{\Delta})$ for the i-th round of Algorithm 3, all vertices after the i-th round have degree at most $\Delta/2^i$ with high probability.*

The next step is to bound the number of steps in each round of Algorithm 3. To do this, Blelloch et al. [BFS12] presented an upper bound on the length of the longest path in the priority DAG induced by the chosen prefix of vertices. Recall that the length of this path provide a bound on the dependence length. The way they do it is by showing that if the subset is not too large, then the longest path in the priority DAG of that subset has length $O(\log n)$.

Theorem 2 *(Rephrasing of Lemma 3.3 of [BFS12]) Assume that all vertices in G have degree at most d. For any $O(\frac{\log n}{d})$-prefix of vertices with respect to a random ordering π, the longest path in the priority DAG of this subset has length $O(\log n)$ with high probability.*

As Theorem 1 provide a bound on the number of rounds of Algorithm 3 and Theorem 2 bounds the number of steps on each of this rounds, we obtain the following.

Theorem 3 *(Rephrasing of Theorem 3.5 of [BFS12]) For a random ordering π of the vertices, the dependence length of the priority DAG is $O(\log \Delta \log n) = O(\log^2 n)$ with high probability. Equivalently, Algorithm 2 requires $O(\log^2 n)$ steps in total when adding the steps on each round.*

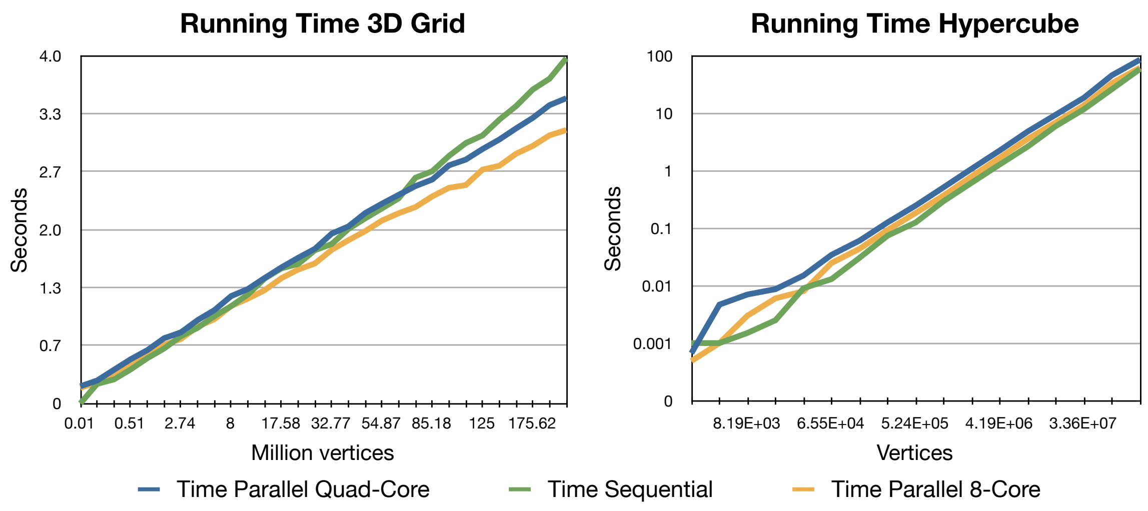


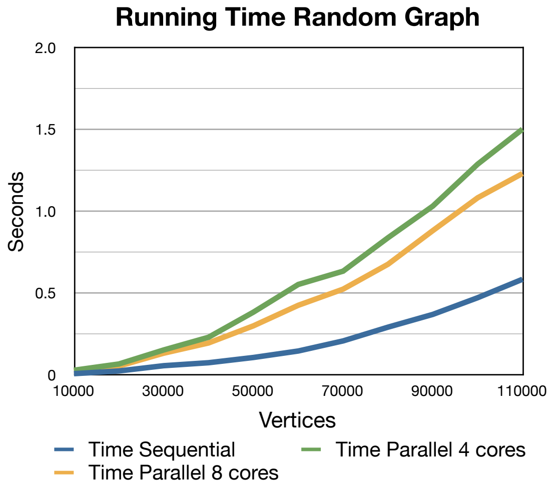
Figure 3: The running time of the sequential algorithm and parallel algorithm in computers with 4 and 8 cores with a 3D grid as input. Parallel algorithm outperforms the sequential algorithm for sufficiently large 3D grids proving that the parallel algorithm is asymptotically faster with as few as 4 processors. For the hypercube, we didn't manage to outperform the sequential algorithm but got a significant improvement when going from 8 to 4 processors (around 1.25 times faster), suggesting that with more processors, we will outperform the sequential algorithm.

4 Results

The objective of this section is to show empirical data that supports the theoretical results presented in the previous sections. Specifically, we implemented parallel Algorithm 2 and counted the number of steps it requires to compute a MIS in different input graphs such as hypercubes, 3D grids and random graphs. These graphs have been used as benchmark for the MIS and related problems [CZF04, BPT11, BFS12]. The idea is that sparse graphs provide the hardest inputs to compute a MIS. Moreover, from the implementation point of view, 3D grids and hypercubes provide an easy way to obtain the set of neighbors of a vertex without storing the whole graph. This allowed us to work with larger inputs that helped us obtain a better view of the asymptotic behavior of the algorithms.

We worked with random graphs, where the probability of the existence of an edge is inversely proportional to the distance between the indices of the vertices. These graphs proved to be hard to work with since the storage in RAM memory of large graphs was not possible. By using main memory, we wouldn't have been able to compare the results with the hypercubes or 3D grids since the impact on performance was significant. Thus, we decided to keep the size of the graph to fit in RAM memory. To construct random graphs, we need to perform a quadratic number of operations (one for every pair of vertices to decide if there is an edge between them). Thus, to deal with vertex sets as large as the ones used in the 3D grids or hypercubes (tenths of million vertices), we would have needed a computation time that we didn't have at hand. Therefore, the size of random graphs was restricted to tenths of thousand vertices. Furthermore, it is known that random graphs are "good" inputs for the sequential algorithm [CRT89], providing a bigger challenge to outperform it using parallelism.

We implemented sequential Algorithm 1 and optimized its performance. We then compared the running time of Algorithms 1 and 2 with the same set of input graphs. We ran



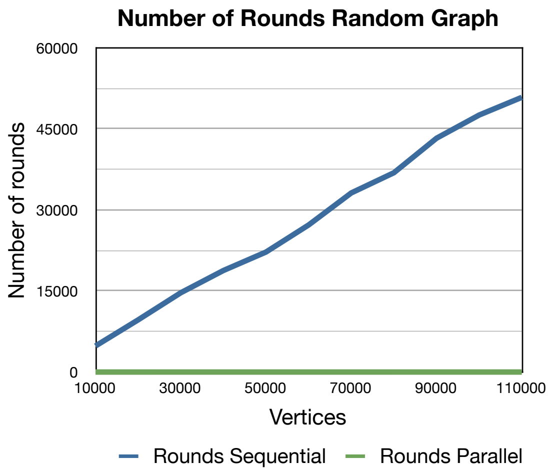


Figure 4: Left: The running time of the sequential and parallel algorithms with random graphs as input. The sequential algorithm is faster than the parallel for small random graphs. As random graphs were proved hard to work with, we have no intuition of the asymptotic behavior. Nevertheless, when going from 4 to 8 processors we obtained a significant improvement suggesting that many processors suffice to outperform the optimized sequential algorithm. Right: The number of rounds with random graphs as input for the sequential and parallel algorithms with a linear and logarithmic behavior, respectively.

the algorithm 30 times for every input size and average the results obtaining in this way significant information on the performance of the algorithm.

Our implementation was done in Cilk in a procedural manner. While Algorithm 2 appears simple at first glance, its implementation requires further details. The main problem is to compute the set of roots on each step of the algorithm. If we use a naïve approach, we will require to check every vertex in the graph to compute the next set of roots. Instead, we look only at a subset of vertices containing all the possible candidates. We maintain the roots of the DAG explicitly in an array, in this way is easy to identify the neighbors of the roots in parallel and remove them from the graph. However, it is trickier to find the new roots afterwards. Notice that when a vertex is processed, it is removed from the graph together with all its neighbors. Therefore, the vertices that can become roots are those lying at distance two from the processed vertices (current roots). By looking only at the neighbors of the neighbors of each processed vertex we can check each candidate and determine which ones belong to the new set of roots. This require extra care since a vertex can be at distance two from several processed vertices. Duplicates can be avoided, however, by having the neighbor write its identifier into the processed vertex using concurrent writing. Whichever write succeeds is responsible for adding the neighbor into the new root array.

The implementation of the algorithm to find new roots has $O(\log n)$ depth using $O(m)$ processors (Lemma 4.1 [BFS12]). Therefore, as the number of steps in Algorithm 2 is at most $O(\log^2 n)$ by Theorem 3, we obtain an implementation having in theory, $O(\log^3 n)$ depth when using $O(m)$ processors.

The overhead of the parallelization is evident as depicted in Fig. 3 where the running time of the parallel algorithm is asymptotically better than that of the sequential algorithm. This can be observed, however, only for graphs larger than 70 million vertices for 4 cores and 11 million vertices for 8 cores suggesting large hidden constants in the big O notation.

Similar results can be observed for the hypercube where we didn't manage to outperform the sequential algorithm, but obtained charts suggesting that, for larger inputs, the parallel algorithm will be able to outperform the sequential with as few as 8 cores. The decrease on the running time when going from 4 to 8 cores provides concise proof that as we utilize more processors, we obtain better performance and outperform the sequential algorithm faster. It is worth noting, however, that Cilk doesn't provide the best platform for this algorithm as a large number of processors are required to outperform the optimized sequential algorithm for small input graphs. Further work could involve implementing this algorithms in MPI (message passing) or GPU (graphics processing unit).

Regardless of the implementation, we provide empirical proof of the intrinsic parallelism of Algorithm 2. In Figures 4 and 5, we observe slow increase in the number of rounds as the size of the graph gets larger. That is, the dependence length of the priority DAG is shallow with almost every random ordering π of the vertices. This provides empirical proofs supporting the theoretical results of Theorem 3. In practice, however, for the hypercube, 3D grid and for random graphs, the number of steps required by Algorithm 2 is in fact logarithmic, suggesting that bounds presented by Blelloch et al. [BFS12] may not be tight.

These results prove that with sufficiently many processors and a good implementation to find new roots, Algorithm 2 should outperform the sequential algorithm.

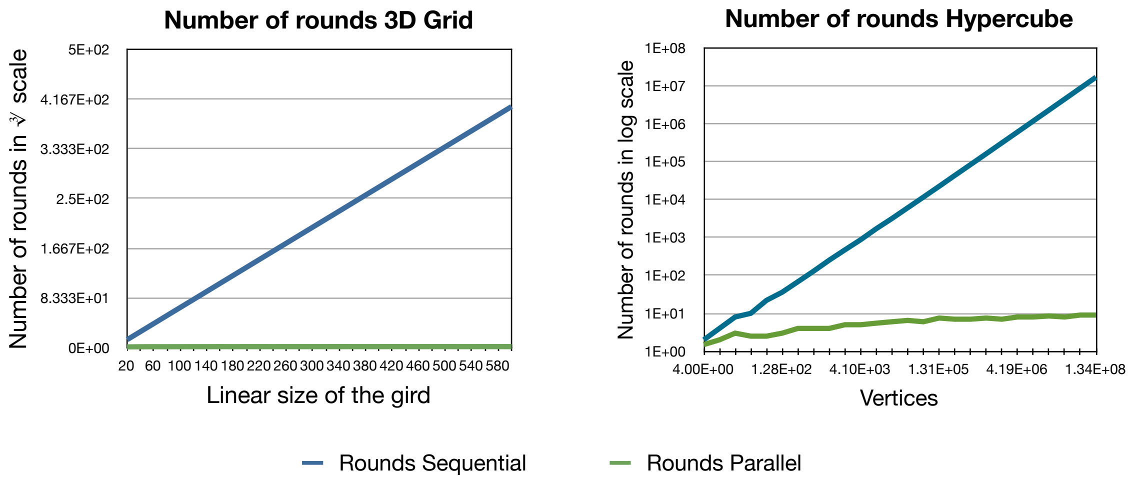


Figure 5: The number of rounds of the sequential and parallel algorithms as a function of the size of the input graph (3D grid and hypercube). In the former algorithm we obtain an almost perfect linear behavior while in the latter, we have a logarithmic behavior.

5 Final remarks

Despite the fact that LFMIS is P-complete, the randomized approach to compute a MIS generates a lot of parallelism. The overhead of the parallel algorithm, however, is huge and requires either for the input size to be large, or to use many processors in order to outperform an optimized sequential algorithm. Also, languages like Cilk may not be optimal for this problem since we need many processors to obtain efficient performance. Hence, the study of this problem in massive parallel computers should be of interest for further research.

References

[ABI86] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7(4):567 – 583, 1986.

[BAAS09] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[BFS12] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 308–317, New York, NY, USA, 2012. ACM.

[BPT11] Guy E. Blelloch, Richard Peng, and Kanat Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 23–32, New York, NY, USA, 2011. ACM.

[Chv79] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. of Oper. Res.*, 4(3):233–235, 1979.

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2 – 22, 1985. International Conference on Foundations of Computation Theory.

[CRT89] Don Coppersmith, Prabhakar Raghavan, and Martin Tompa. Parallei graph algorithms that are efficient on average. *Information and Computation*, 81(3):318 – 333, 1989.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[EH00] Lars Engebretsen and Jonas Holmerin. Clique is hard to approximate within $n^{1-o(1)}$. In Ugo Montanari, José Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 2–12. Springer Berlin / Heidelberg, 2000.

[GS89] Mark Goldberg and Thomas Spencer. A new parallel algorithm for the maximal independent set problem, 1989.

[KW85] Richard M. Karp and Avi Wigderson. A fast parallel algorithm for the maximal independent set problem. *J. ACM*, 32(4):762–773, October 1985.

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[Val82] L. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.